# Contents

## 1 Linear Systems
- Matrix Class ................................................................. 1
- ComplexMatrix Class ......................................................... 3
- LU Class .............................................................................. 6
- ComplexLU Class ................................................................. 10
- Cholesky Class .................................................................... 13
- QR Class .............................................................................. 16
- SVD Class ............................................................................. 20

## 2 Eigensystem Analysis
- Eigen Class ........................................................................... 23
- SymEigen Class ..................................................................... 27

## 3 Interpolation and Approximation
- Spline Class .......................................................................... 29
- CsAkima Class ....................................................................... 31
- CsInterpolate Class ............................................................... 35
- CsInterpolate.Condition Enumeration .................................. 37
- CsPeriodic Class .................................................................... 39
- CsShape Class ........................................................................ 41
- CsSmooth Class .................................................................... 44
- CsSmoothC2 Class ................................................................. 47
- BSpline Class ......................................................................... 50

---

Contents          Contents • i
9 Special Functions

Sfun Class ............................................. 171
Bessel Class ........................................... 187

10 Miscellaneous

Complex Structure .................................... 193
Physical Structure .................................... 211
EpsilonAlgorithm Class ............................... 222

11 Printing Functions

PrintMatrix Class ..................................... 225
PrintMatrixFormat Class ............................. 229
PrintMatrixFormat.ParsePosition Class ......... 232
PrintMatrix.MatrixType Enumeration ............. 232
12 Basic Statistics

Summary Class .................................................. 238
Covariances Class ................................................. 246
Covariances.MatrixType Enumeration .......................... 251
NormOneSample Class ............................................ 251
NormTwoSample Class ............................................ 256
Sort Class ........................................................ 265
Ranks Class ....................................................... 272
Ranks.Tie Enumeration .......................................... 279
TableOneWay Class .............................................. 279
TableTwoWay Class ............................................... 284
TableMultiWay Class ............................................ 288
TableMultiWay.TableBalanced Class ......................... 294
TableMultiWay.TableUnbalanced Class ....................... 295

13 Regression

LinearRegression Class .......................................... 297
LinearRegression.CoefficientTTestsValue Class .......... 301
NonlinearRegression Class ..................................... 302
NonlinearRegression.IDerivative Interface .................... 314
NonlinearRegression.IFunction Interface ..................... 315
SelectionRegression Class ..................................... 315
SelectionRegression.Criterion Enumeration .................. 326
SelectionRegression.SummaryStatistics Class .............. 326
StepwiseRegression Class ...................................... 327
StepwiseRegression.CoefficientTTestsValue Class ........ 335
StepwiseRegression.Direction Enumeration ................... 336
UserBasisRegression Class ..................................... 337
InputLayer Class ............................................. 700
HiddenLayer Class ........................................... 701
OutputLayer Class ........................................... 702
Node Class .................................................... 704
InputNode Class ............................................. 705
Perceptron Class ............................................. 706
OutputPerceptron Class ..................................... 707
IActivation Interface ...................................... 708
Link Class .................................................... 709
ITrainer Interface .......................................... 710
QuasiNewtonTrainer Class ................................. 711
QuasiNewtonTrainer.IError Interface ................. 715
LeastSquaresTrainer Class ............................... 716
EpochTrainer Class ......................................... 719
ScaleFilter Class ............................................ 721
ScaleFilter.ScalingMethod Enumeration .............. 729
UnsupervisedNominalFilter Class ....................... 730
UnsupervisedOrdinalFilter Class ....................... 733
UnsupervisedOrdinalFilter.TransformMethod Enumeration .................. 738
TimeSeriesFilter Class ..................................... 738
TimeSeriesClassFilter Class ............................. 741
Network Class ............................................... 762

24 Miscellaneous ........................................... 771
Warning Class ............................................... 771
WarningObject Class ....................................... 772
IMSLException Class ....................................... 773

25 Exceptions ................................................. 777
BadInitialGuessException Class ......................... 779
BoundsInconsistentException Class .................... 781
ConstraintEvaluationException Class ................... 783
<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstraintsInconsistentException Class</td>
<td>785</td>
</tr>
<tr>
<td>ConstraintsNotSatisfiedException Class</td>
<td>787</td>
</tr>
<tr>
<td>DidNotConvergeException Class</td>
<td>789</td>
</tr>
<tr>
<td>EqualityConstraintsException Class</td>
<td>791</td>
</tr>
<tr>
<td>FalseConvergenceException Class</td>
<td>793</td>
</tr>
<tr>
<td>IllConditionedException Class</td>
<td>795</td>
</tr>
<tr>
<td>InconsistentSystemException Class</td>
<td>797</td>
</tr>
<tr>
<td>LimitingAccuracyException Class</td>
<td>799</td>
</tr>
<tr>
<td>LinearlyDependentGradientsException Class</td>
<td>801</td>
</tr>
<tr>
<td>MaxIterationsException Class</td>
<td>803</td>
</tr>
<tr>
<td>MaxNumberStepsNotAllowedException Class</td>
<td>805</td>
</tr>
<tr>
<td>NoAcceptableStepsSizeException Class</td>
<td>807</td>
</tr>
<tr>
<td>NotSPDException Class</td>
<td>809</td>
</tr>
<tr>
<td>NumericDifficultyException Class</td>
<td>811</td>
</tr>
<tr>
<td>ObjectiveEvaluationException Class</td>
<td>813</td>
</tr>
<tr>
<td>PenaltyFunctionPointInfeasibleException Class</td>
<td>815</td>
</tr>
<tr>
<td>ProblemInfeasibleException Class</td>
<td>817</td>
</tr>
<tr>
<td>ProblemUnboundedException Class</td>
<td>819</td>
</tr>
<tr>
<td>QPInfeasibleException Class</td>
<td>821</td>
</tr>
<tr>
<td>SingularException Class</td>
<td>823</td>
</tr>
<tr>
<td>SingularMatrixException Class</td>
<td>825</td>
</tr>
<tr>
<td>TerminationCriteriaNotSatisfiedException Class</td>
<td>827</td>
</tr>
<tr>
<td>ToleranceTooSmallException Class</td>
<td>829</td>
</tr>
<tr>
<td>TooManyIterationsException Class</td>
<td>832</td>
</tr>
<tr>
<td>UnboundedBelowException Class</td>
<td>834</td>
</tr>
<tr>
<td>VarBoundsInconsistentException Class</td>
<td>836</td>
</tr>
<tr>
<td>WorkingSetSingularException Class</td>
<td>838</td>
</tr>
<tr>
<td>AllDeletedException Class</td>
<td>840</td>
</tr>
<tr>
<td>AllMissingException Class</td>
<td>842</td>
</tr>
<tr>
<td>BadVarianceException Class</td>
<td>844</td>
</tr>
<tr>
<td>ClassificationVariableException Class</td>
<td>846</td>
</tr>
</tbody>
</table>
ClassificationVariableLimitException Class ........................................ 848
ClassificationVariableValueException Class .................................... 850
ClusterNoPointsException Class ...................................................... 852
ConstrInconsistentException Class .................................................. 854
CovarianceSingularException Class .................................................. 856
CyclingIsOccurringException Class .................................................. 858
DeleteObservationsException Class .................................................. 860
DidNotConvergeException Class ...................................................... 862
DiffObsDeletedException Class ....................................................... 864
EigenvalueException Class ............................................................. 866
EmptyGroupException Class ............................................................ 868
EqConstrInconsistentException Class ............................................... 870
IllConditionedException Class ......................................................... 872
IncreaseErrRelException Class ......................................................... 874
MatrixSingularException Class ........................................................ 876
MoreObsDelThanEnteredException Class .......................................... 878
NegativeFreqException Class .......................................................... 881
NegativeWeightException Class ....................................................... 883
NewInitialGuessException Class ....................................................... 885
NoConvergenceException Class ....................................................... 887
NoDegreesOfFreedomException Class ............................................... 889
NoVariationInputException Class ..................................................... 891
NoVectorXException Class ............................................................... 893
NonPosVarianceException Class ....................................................... 895
NonPosVarianceXYException Class ................................................... 897
NonPositiveEigenvalueException Class ............................................. 899
NonPositiveVarianceException Class ................................................. 901
NotCDFException Class ................................................................. 903
NotPositiveDefiniteException Class ............................................... 906
NotPositiveSemiDefiniteException Class ......................................... 908
NotSemiDefiniteException Class ...................................................... 910
NoVariablesEnteredException Class .................................... 912
NoVariablesException Class ........................................... 914
NoVariationInputException Class ..................................... 916
NoVectorXException Class .............................................. 918
PooledCovarianceSingularException Class .............................. 920
RankException Class ..................................................... 922
ScaleFactorZeroException Class ....................................... 924
SingularException Class .................................................. 926
SumOfWeightsNegException Class ....................................... 928
TooManyCallsException Class .......................................... 930
TooManyFunctionEvaluationsException Class .......................... 932
TooManyIterationsException Class ...................................... 934
TooManyJacobianEvalException Class .................................. 936
TooManyObsDeletedException Class ................................... 938
VarsDeterminedException Class .......................................... 940
ZeroNormException Class .................................................. 942

26 References .................................................................. 945
Chapter 1: Linear Systems

Types

- class Matrix ................................................................. 3
- class ComplexMatrix ......................................................... 6
- class LU ................................................................. 10
- class ComplexLU ....................................................... 13
- class Cholesky ......................................................... 16
- class QR ............................................................. 20
- class SVD ............................................................. 23

Usage Notes

Solving Systems of Linear Equations

A square system of linear equations has the form $Ax = b$, where $A$ is a user-specified $n \times n$ matrix, $b$ is a given right-hand side $n$ vector, and $x$ is the solution $n$ vector. Each entry of $A$ and $b$ must be specified by the user. The entire vector $x$ is returned as output.

When $A$ is invertible, a unique solution to $Ax = b$ exists. The most commonly used direct method for solving $Ax = b$ factors the matrix $A$ into a product of triangular matrices and solves the resulting triangular systems of linear equations. Functions that use direct methods for solving systems of linear equations all compute the solution to $Ax = b$.

Matrix Factorizations

In some applications, it is desirable to just factor the $n \times n$ matrix $A$ into a product of two triangular matrices. This can be done by a constructor of a class for solving the system of linear equations $Ax = b$. The constructor of class LU computes the LU factorization of $A$.

Besides the basic matrix factorizations, such as $LU$ and $LL^T$, additional matrix factorizations also are provided. For a real matrix $A$, its $QR$ factorization can be computed using the class QR. The class for computing the singular value decomposition (SVD) of a matrix is discussed in a
Matrix Inversions

The inverse of an $n \times n$ nonsingular matrix can be obtained by using the method Inverse in the classes for solving systems of linear equations. The inverse of a matrix need not be computed if the purpose is to solve one or more systems of linear equations. Even with multiple right-hand sides, solving a system of linear equations by computing the inverse and performing matrix multiplication is usually more expensive than the method discussed in the next section.

Multiple Right-Hand Sides

Consider the case where a system of linear equations has more than one right-hand side vector. It is most economical to find the solution vectors by first factoring the coefficient matrix $A$ into products of triangular matrices. Then, the resulting triangular systems of linear equations are solved for each right-hand side. When $A$ is a real general matrix, access to the LU factorization of $A$ is computed by a constructor of LU. The solution $x_k$ for the $k$-th right-hand side vector, $b_k$ is then found by two triangular solves, $Ly_k = b_k$ and $Ux_k = y_k$. The method Solve in class LU is used to solve each right-hand side. These arguments are found in other functions for solving systems of linear equations.

Least-Squares Solutions and QR Factorizations

Least-squares solutions are usually computed for an over-determined system of linear equations $A_{m \times n}x = b$, where $m > n$. A least-squares solution $x$ minimizes the Euclidean length of the residual vector $r = Ax - b$. The class QR computes a unique least-squares solution for $x$ when $A$ has full column rank. If $A$ is rank-deficient, then the base solution for some variables is computed. These variables consist of the resulting columns after the interchanges. The QR decomposition, with column interchanges or pivoting, is computed such that $AP = QR$. Here, $Q$ is orthogonal, $R$ is upper-trapezoidal with its diagonal elements nonincreasing in magnitude, and $P$ is the permutation matrix determined by the pivoting. The base solution $x_B$ is obtained by solving $R(P^T)x = Q^Tb$ for the base variables. For details, see class QR. The QR factorization of a matrix $A$ such that $AP = QR$ with $P$ specified by the user can be computed using keywords.

Singular Value Decompositions and Generalized Inverses

The SVD of an $m \times n$ matrix $A$ is a matrix decomposition $A = USV^T$. With $q = \min(m, n)$, the factors $U_{m \times q}$ and $V_{n \times q}$ are orthogonal matrices, and $S_{q \times q}$ is a nonnegative diagonal matrix with nonincreasing diagonal terms. The class SVD computes the singular values of $A$ by default. Part or all of the $U$ and $V$ matrices, an estimate of the rank of $A$, and the generalized inverse of $A$, also can be obtained.
Ill-Conditioning and Singularity

An \( m \times n \) matrix \( A \), is mathematically singular if there is an \( x \neq 0 \) such that \( Ax = 0 \). In this case, the system of linear equations \( Ax = b \) does not have a unique solution. On the other hand, a matrix \( A \) is numerically singular if it is "close" to a mathematically singular matrix. Such problems are called ill-conditioned. If the numerical results with an ill-conditioned problem are unacceptable, users can obtain an approximate solution to the system using the SVD of \( A \): If \( q = \min(m, n) \) and

\[
A = \sum_{i=1}^{q} s_{i,i} u_i v_i^T
\]

then the approximate solution is given by the following:

\[
x_k = \sum_{i=1}^{k} t_{i,i} (b^T u_i) v_i
\]

The scalars \( t_{i,i} \) are defined below.

\[
t_{i,i} = \begin{cases} 
  s_{i,i}^{-1} & \text{if } s_{i,i} \geq \text{tol} > 0 \\
  0 & \text{otherwise}
\end{cases}
\]

The user specifies the value of \( \text{tol} \). This value determines how "close" the given matrix is to a singular matrix. Further restrictions may apply to the number of terms in the sum, \( k \leq q \). For example, there may be a value of \( k \leq q \) such that the scalars \( |b^T u_i|, i > k \) are smaller than the average uncertainty in the right-hand side \( b \). This means that these scalars can be replaced by zero; and hence, \( b \) is replaced by a vector that is within the stated uncertainty of the problem.

Matrix Class

Matrix manipulation functions.

public class Imsl.Math.Matrix

Methods

Add

static public double[,] Add(double[,] a, double[,] b)
Add two rectangular matrices, \( a + b \).

a A double rectangular matrix.
b A double rectangular matrix.
Returns — A double rectangular matrix representing the matrix sum of the two arguments.

System.ArgumentException is thrown when the matrices are not the same size

CheckSquareMatrix

static public void CheckSquareMatrix(double[,] a)

Check that the matrix is square.

a A double matrix.

System.ArgumentException is thrown when the matrix is not square

FrobeniusNorm

static public double FrobeniusNorm(double[,] a)

Return the frobenius norm of a matrix.

a A double rectangular matrix.

Returns — A double scalar value equal to the frobenius norm of the matrix.

InfinityNorm

static public double InfinityNorm(double[,] a)

Return the infinity norm of a matrix.

a A double rectangular matrix.

Returns — A double scalar value equal to the maximum of the row sums of the absolute values of the matrix elements.

Multiply

static public double[] Multiply(double[] x, double[,] a)

Return the product of the row matrix x and the rectangular matrix a.

x A double row matrix.

a A double rectangular matrix.

Returns — A double vector representing the product of the arguments, x*a.

System.ArgumentException is thrown when the number of elements in the input row matrix is not equal to the number of rows of the matrix

Multiply

static public double[] Multiply(double[,] a, double[] x)

Multiply the rectangular matrix a and the column matrix x.

a A double rectangular matrix.

x A double column matrix.

Returns — A double vector representing the product of the arguments, a * x.
System.ArgumentException is thrown when the number of columns in the input matrix is not equal to the number of elements in the input column vector.

Multiply

static public double[,] Multiply(double[,] a, double[,] b)
Multiply two rectangular matrices, a * b.

a  A double rectangular matrix.
b  A double rectangular matrix.

Returns — The double matrix product of a * b.

System.ArgumentException is thrown when the number of columns in a is not equal to the number of rows in b.

OneNorm

static public double OneNorm(double[,] a)
Return the matrix one norm.

a  A double rectangular matrix.

Returns — A double value equal to the maximum of the column sums of the absolute values of the matrix elements.

Subtract

static public double[,] Subtract(double[,] a, double[,] b)
Subtract two rectangular matrices, a - b.

a  A double rectangular matrix.
b  A double rectangular matrix.

Returns — A double rectangular matrix representing the matrix difference of the two arguments.

System.ArgumentException is thrown when the matrices are not the same size.

Transpose

static public double[,] Transpose(double[,] a)
Return the transpose of a matrix.

a  A double matrix.

Returns — A double matrix which is the transpose of the argument.
Example: Matrix and PrintMatrix

The 1 norm of a matrix is found using a method from the Matrix class. The matrix is printed using the PrintMatrix class.

```csharp
using System;
using Imsl.Math;

class MatrixEx1
{
    public static void Main(string[] args)
    {
        double nrm1;
        double[,] a = {
           {0.0, 1.0, 2.0, 3.0},
           {4.0, 5.0, 6.0, 7.0},
           {8.0, 9.0, 8.0, 1.0},
           {6.0, 3.0, 4.0, 3.0}
        };

        // Get the 1 norm of matrix a
        nrm1 = Matrix.OneNorm(a);

        // Construct a PrintMatrix object with a title
        PrintMatrix p = new PrintMatrix("A Simple Matrix");

        // Print the matrix and its 1 norm
        p.Print(a);
        Console.Out.WriteLine("The 1 norm of the matrix is " + nrm1);
    }
}
```

Output

```
A Simple Matrix
0 1 2 3
0 0 1 2 3
1 4 5 6 7
2 8 9 8 1
3 6 3 4 3

The 1 norm of the matrix is 20
```

ComplexMatrix Class

Complex matrix manipulation functions.
public class Imsl.Math.ComplexMatrix

Methods

Add
Add two rectangular Complex arrays, a + b.

a A Complex rectangular array.
b A Complex rectangular array.

Returns — The Complex matrix sum of the two arguments

System.ArgumentException is thrown when the matricies are not the same size

CheckSquareMatrix
static public void CheckSquareMatrix(Imsl.Math.Complex[,] a)
Check that the Complex matrix is square.
a A Complex matrix.

System.ArgumentException is thrown when the matrix is not square

FrobeniusNorm
static public double FrobeniusNorm(Imsl.Math.Complex[,] a)
Return the frobenius norm of a Complex matrix.
a A Complex rectangular matrix.

Returns — A double value equal to the frobenius norm of the matrix.

InfinityNorm
static public double InfinityNorm(Imsl.Math.Complex[,] a)
Return the infinity norm of a Complex matrix.
a A Complex rectangular matrix.

Returns — A double value equal to the maximum of the row sums of the absolute values of the array elements.

Multiply
Returns the product of the row vector x and the rectangular array a, both Complex.
x A Complex row vector.
a A Complex rectangular matrix.
Returns — A Complex vector containing the product of the arguments, $x \ast a$.

[System.ArgumentException] is thrown when the number of elements in the input vector is not equal to the number of rows of the matrix.

### Multiply

```csharp
```

Multiply the rectangular array $a$ and the column vector $x$, both Complex.

- $a$: A Complex rectangular matrix.
- $x$: A Complex vector.

Returns — A Complex vector containing the product of the arguments, $a \ast x$.

[System.ArgumentException] is thrown when the number of columns in the input matrix is not equal to the number of elements in the input vector.

### Multiply

```csharp
```

Multiply two Complex rectangular arrays, $a \ast b$.

- $a$: A Complex rectangular array.
- $b$: A Complex rectangular array.

Returns — The Complex matrix product of $a$ times $b$.

[System.ArgumentException] is thrown when the number of columns in $a$ is not equal to the number of rows in $b$.

### OneNorm

```csharp
static public double OneNorm(Imsl.Math.Complex[,] a)
```

Return the Complex matrix one norm.

- $a$: A Complex rectangular array.

Returns — A double value equal to the maximum of the column sums of the absolute values of the array elements.

### Subtract

```csharp
```

Subtract two Complex rectangular arrays, $a - b$.

- $a$: A Complex rectangular array.
- $b$: A Complex rectangular array.

Returns — The Complex matrix difference of the two arguments.
System.ArgumentException is thrown when the matrices are not the same size.

**Transpose**

```csharp
Return the transpose of a Complex matrix.
a A Complex matrix.
Returns — The Complex matrix transpose of the argument.
```

### Example: Print a Complex Matrix

A Complex matrix and its transpose is printed.

```csharp
using System;
using Imsl.Math;

public class ComplexMatrixEx1
{
    public static void Main(String[] args)
    {
        Complex[,] a = {
           {new Complex(1, 3), new Complex(3, 5), new Complex(7, 9)},
           {new Complex(8, 7), new Complex(9, 5), new Complex(1, 9)},
           {new Complex(2, 9), new Complex(6, 9), new Complex(7, 3)},
           {new Complex(5, 4), new Complex(8, 4), new Complex(5, 9)}
        };

        // Print the matrix
        new PrintMatrix("Matrix A").Print(a);

        // Print the transpose of the matrix
        new PrintMatrix("Transpose(A)").Print(ComplexMatrix.Transpose(a));
    }
}
```

**Output**

Matrix A

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1+3i</td>
<td>3+5i</td>
</tr>
<tr>
<td>1</td>
<td>8+7i</td>
<td>9+5i</td>
</tr>
<tr>
<td>2</td>
<td>2+9i</td>
<td>6+9i</td>
</tr>
<tr>
<td>3</td>
<td>5+4i</td>
<td>8+4i</td>
</tr>
</tbody>
</table>

Transpose(A)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1+3i</td>
<td>8+7i</td>
<td>2+9i</td>
</tr>
<tr>
<td>1</td>
<td>3+5i</td>
<td>9+5i</td>
<td>6+9i</td>
</tr>
<tr>
<td>2</td>
<td>7+9i</td>
<td>1+9i</td>
<td>7+3i</td>
</tr>
</tbody>
</table>
LU Class

LU factorization of a matrix of type double.

public class Imsl.Math.LU

Constructor

LU

public LU(double[,] a)

Creates the LU factorization of a square matrix of type double.

a  The double square matrix to be factored.

Imsl.Math.SingularMatrixException is thrown when the input matrix is singular

Methods

Condition

public double Condition(double[,] a)

Return an estimate of the reciprocal of the L1 condition number of a matrix.

a  The double square matrix for which the reciprocal of the L1 condition number is desired.

Returns — A double value representing an estimate of the reciprocal of the L1 condition number of the matrix.

Determinant

public double Determinant()

Return the determinant of the matrix used to construct this instance.

Returns — A double scalar containing the determinant of the matrix used to construct this instance.

Inverse

public double[,] Inverse()

Returns the inverse of the matrix used to construct this instance.

Returns — A double matrix representing the inverse of the matrix used to construct this instance.

Solve

public double[] Solve(double[] b)

Solve ax=b for x using the LU factorization of a.

b  A double array containing the right-hand side of the linear system.
Returns — A double array containing the solution to the linear system of equations.

Solve

```java
static public double[] Solve(double[,] a, double[] b)
```
Solve ax=b for x using the LU factorization of a.

- `a` — A double square matrix.
- `b` — A double column vector.

Returns — A double column vector containing the solution to the linear system of equations.

**System.ArgumentException** is thrown when the number of rows in the input matrix is not equal to the number of elements in x

**Imsl.Math.SingularMatrixException** is thrown when the matrix is singular.

SolveTranspose

```java
public double[] SolveTranspose(double[] b)
```
Return the solution x of the linear system transpose(A)x = b.

- `b` — A double array containing the right-hand side of the linear system.

Returns — A double array containing the solution to the linear system of equations.

**Description**

LU performs an LU factorization of a real general coefficient matrix. The **Condition** method estimates the condition number of the matrix. The LU factorization is done using scaled partial pivoting. Scaled partial pivoting differs from partial pivoting in that the pivoting strategy is the same as if each row were scaled to have the same infinity norm.

The $L_1$ condition number of the matrix $A$ is defined to be $\kappa(A) = ||A||_1||A^{-1}||_1$. Since it is expensive to compute $||A^{-1}||_1$, the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described in a paper by Cline et al. (1979).

An estimated condition number greater than $1/\epsilon$ (where $\epsilon$ is machine precision) indicates that very small changes in $A$ can cause very large changes in the solution $x$. Iterative refinement can sometimes find the solution to such a system.

LU fails if $U$, the upper triangular part of the factorization, has a zero diagonal element. This can occur only if $A$ either is singular or is very close to a singular matrix.

Use the **Solve** method to solve systems of equations. The **Determinant** method can be called to compute the determinant of the coefficient matrix.

LU is based on the LINPACK routine SGECO; see Dongarra et al. (1979). SGECO uses unscaled partial pivoting.
Example: LU Factorization of a Matrix

The LU Factorization of a Matrix is performed. A linear system is then solved using the factorization. The inverse, determinant, and condition number of the input matrix are also computed.

```csharp
using System;
using Imsl.Math;

class LUEx1
{
    public static void Main(String[] args)
    {
        double[,] a = {
            {1, 3, 3},
            {1, 3, 4},
            {1, 4, 3}
        };
        double[] b = new double[]{12, 13, 14};

        // Compute the LU factorization of A
        LU lu = new LU(a);

        // Solve Ax = b
        double[] x = lu.Solve(b);
        new PrintMatrix("x").Print(x);

        // Find the inverse of A.
        double[,] ainv = lu.Inverse();
        new PrintMatrix("ainv").Print(ainv);

        // Find the condition number of A.
        double condition = lu.Condition(a);
        Console.Out.WriteLine("condition number = " + condition);

        // Find the determinant of A.
        double determinant = lu.Determinant();
        Console.Out.WriteLine("determinant = " + determinant);
    }
}
```

Output

```
x
0
0 3
1 2
2 1

ainv
0 1 2
```

IMSL C# Numerical Library
0    7   -3   -3
1    -1  2.22044604925031E-16  1
2    -1   1     0

condition number = 0.0151202749140893
determinant = -1

ComplexLU Class

LU factorization of a matrix of type Complex.

public class Imsl.Math.ComplexLU

Constructor

ComplexLU
    public ComplexLU(Imsl.Math.Complex[,] a)
    Creates the LU factorization of a square matrix of type Complex.

        a  The Complex square matrix to be factored.

Imsl.Math.SingularMatrixException is thrown when the input matrix is singular

Methods

Condition
    public double Condition(Imsl.Math.Complex[,] a)
    Return an estimate of the reciprocal of the L1 condition number.

        a  A Complex matrix.

    Returns — A double scalar value representing the estimate of the reciprocal of the L1
    condition number of the matrix a.

Determinant
    public Imsl.Math.Complex Determinant()
    Return the determinant of the matrix used to construct this instance.

    Returns — A Complex scalar containing the determinant of the matrix used to construct
    this instance.

Inverse
    public Imsl.Math.Complex[,] Inverse()
    Compute the inverse of a matrix of type Complex.
Returns — A Complex matrix containing the inverse of the matrix used to construct this object.

Solve

```csharp
Return the solution x of the linear system ax = b using the LU factorization of a.
```

b  A Complex array containing the right-hand side of the linear system.

Returns — A Complex array containing the solution to the linear system of equations.

```csharp
Return the solution x of the linear system ax = b using the LU factorization of a.
```

a  A Complex square matrix.

b  A Complex column vector.

Returns — A Complex column vector containing the solution to the linear system of equations.

System.ArgumentException is thrown when the number of rows in a is not equal to the length of b

Imsl.Math.SingularMatrixException is thrown when the matrix is singular

SolveTranspose

```csharp
Return the solution x of the linear system transpose(A)x = b.
```

b  A Complex array containing the right-hand side of the linear system.

Returns — A Complex array containing the solution to the linear system of equations.

Description

ComplexLU performs an LU factorization of a complex general coefficient matrix. ComplexLU’s method Condition estimates the condition number of the matrix. The LU factorization is done using scaled partial pivoting. Scaled partial pivoting differs from partial pivoting in that the pivoting strategy is the same as if each row were scaled to have the same infinity norm.

The $L_1$ condition number of the matrix $A$ is defined to be $\kappa (A) = \|A\|_1 \|A^{-1}\|_1$. Since it is expensive to compute $\|A^{-1}\|_1$, the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described by Cline et al. (1979).

An estimated condition number greater than $1/\epsilon$ (where $\epsilon$ is machine precision) indicates that very small changes in $A$ can cause very large changes in the solution $x$. Iterative refinement can sometimes find the solution to such a system.
ComplexLU fails if $U$, the upper triangular part of the factorization, has a zero diagonal element. This can occur only if $A$ either is singular or is very close to a singular matrix.

The Solve method can be used to solve systems of equations. The method Determinant can be called to compute the determinant of the coefficient matrix.

ComplexLU is based on the LINPACK routine CGECO; see Dongarra et al. (1979). CGECO uses unscaled partial pivoting.

**Example: LU Decomposition of a Complex Matrix**

The Complex class is used to convert a real matrix to a Complex matrix. An LU decomposition of the matrix is performed and the determinant and condition number of the matrix are obtained.

```csharp
using System;
using Imsl.Math;

public class ComplexLUEx1
{
    public static void Main(String[] args)
    {
        double[,] ar = {{1, 3, 3},
                        {1, 3, 4},
                        {1, 4, 3}};
        double[] br = {12, 13, 14};

        Complex[,] a = new Complex[3,3];
        Complex[] b = new Complex[3];

        for (int i = 0; i < 3; i++)
        {
            b[i] = new Complex(br[i]);
            for (int j = 0; j < 3; j++)
            {
                a[i,j] = new Complex(ar[i,j]);
            }
        }

        // Compute the LU factorization of A
        ComplexLU clu = new ComplexLU(a);

        // Solve Ax = b
        Complex[] x = clu.Solve(b);
        Console.Out.WriteLine("The solution is:");
        new PrintMatrix("x").Print(x);

        // Find the condition number of A.
        double condition = clu.Condition(a);
        Console.Out.WriteLine("The condition number = "+ condition);
    }
}
```

Linear Systems

ComplexLU Class • 15
// Find the determinant of A.
Complex determinant = clu.Determinant();
Console.Out.WriteLine("The determinant = " + determinant);
}
}

Output

The solution is:

```
x
0
0 3
1 2
2 1
```

The condition number = 0.0148867313915858

The determinant = -1

---

**Cholesky Class**

Cholesky factorization of a matrix of type `double`.

```csharp
public class Imsl.Math.Cholesky
```

**Constructor**

```
public Cholesky(double[,] a)
Create the Cholesky factorization of a symmetric positive definite matrix of type double.
```

- `a` - A double square matrix to be factored.

**Imsl.Math.SingularMatrixException** is thrown when the input matrix `a` is singular

**Imsl.Math.NotSPDException** is thrown when the input matrix is not symmetric, positive definite.

**Methods**

```
public void Downdate(double[] x)
```
Downdates the factorization by subtracting a rank-1 matrix.
The object will contain the Cholesky factorization of a - x * transpose(x), where a is the
previously factor matrix.

x A double array which specifies the rank-1 matrix. x is not modified by this function.

Imsl.Math.NotSPDEException is thrown if a - x * transpose(x) is not symmetric
positive-definite.

GetR
public double[,] GetR()
The R matrix that results from the Cholesky factorization.
R is a lower triangular matrix and A = RR^T.
Returns — A double matrix which contains the R matrix that results from the Cholesky
factorization.

Inverse
public double[,] Inverse()
Returns the inverse of this matrix.
Returns — A double matrix containing the inverse.

Solve
public double[] Solve(double[] b)
Solve Ax = b where A is a positive definite matrix with elements of type double.
b A double array containing the right-hand side of the linear system.
Returns — A double array containing the solution to the system of linear equations.

Update
public void Update(double[] x)
Updates the factorization by adding a rank-1 matrix.
The object will contain the Cholesky factorization of a + x * transpose(x), where a is the
previously factored matrix.

x A double array which specifies the rank-1 matrix. x is not modified by this function.

Description
Class Cholesky is based on the LINPACK routine SCHDC; see Dongarra et al. (1979).
Before the decomposition is computed, initial elements are moved to the leading part of A and
final elements to the trailing part of A. During the decomposition only rows and columns
corresponding to the free elements are moved. The result of the decomposition is an upper
triangular matrix R and a permutation matrix P that satisfy P^TAP = R^T R, where P is
represented by ipvt.
The method Update is based on the LINPACK routine SCHUD; see Dongarra et al. (1979).
The Cholesky factorization of a matrix is \( A = R^T R \), where \( R \) is an upper triangular matrix. Given this factorization, \textbf{Downate} computes the factorization \( A - xx^T = \tilde{R}^T \tilde{R} \).

\textbf{Downate} determines an orthogonal matrix \( U \) as the product \( G_N \ldots G_1 \) of Givens rotations, such that

\[
U \begin{bmatrix} R \\ 0 \end{bmatrix} = \begin{bmatrix} \tilde{R} \\ x^T \end{bmatrix}
\]

By multiplying this equation by its transpose and noting that \( U^T U = I \), the desired result

\[
R^T R - xx^T = \tilde{R}^T \tilde{R}
\]

is obtained.

Let \( a \) be the solution of the linear system \( R^T a = x \) and let

\[
\alpha = \sqrt{1 - \|a\|^2_2}
\]

The Givens rotations, \( G_i \), are chosen such that

\[
G_1 \ldots G_N \begin{bmatrix} a \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}
\]

The \( G_i \) are \((N + 1) \times (N + 1)\) matrices of the form

\[
G_i = \begin{bmatrix}
I_{i-1} & 0 & 0 & 0 \\
0 & c_i & 0 & -s_i \\
0 & 0 & I_{N-i} & 0 \\
0 & s_i & 0 & c_i
\end{bmatrix}
\]

where \( I_k \) is the identity matrix of order \( k \); and \( c_i = \cos \theta_i, s_i = \sin \theta_i \) for some \( \theta_i \).

The Givens rotations are then used to form

\[
\tilde{R}, G_1 \ldots G_N \begin{bmatrix} R \\ 0 \end{bmatrix} = \begin{bmatrix} \tilde{R} \\ x^T \end{bmatrix}
\]

The matrix

\( \tilde{R} \)

is upper triangular and

\( \tilde{x} = x \)

because

\[
x = (R^T 0) \begin{bmatrix} a \\ \alpha \end{bmatrix} = (R^T 0) U^T U \begin{bmatrix} a \\ \alpha \end{bmatrix} = (\tilde{R}^T \tilde{x}) \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \tilde{x}
\]
Example: Cholesky Factorization

The Cholesky Factorization of a matrix is performed as well as its inverse.

```csharp
using System;
using Imsl.Math;

public class CholeskyEx1
{
    public static void Main(String[] args)
    {
        double[,] a = {
            {1, -3, 2},
            {-3, 10, -5},
            {2, -5, 6}
        };
        double[] b = new double[]{27, -78, 64};

        // Compute the Cholesky factorization of A
        Cholesky cholesky = new Cholesky(a);

        // Solve Ax = b
        double[] x = cholesky.Solve(b);
        new PrintMatrix("x").Print(x);

        // Find the inverse of A.
        double[,] ainv = cholesky.Inverse();
        new PrintMatrix("ainv").Print(ainv);
    }
}
```

Output

```
x
  0
  0 1
  1 -4
  2 7

  ainv
  0 1 2
  0 35 8 -5
  1 8 2 -1
  2 -5 -1 1
```

Linear Systems    Cholesky Class • 19
QR Class

QR Decomposition of a matrix.

class Imsl.Math.QR

Constructor

QR

public QR(double[,] a)

Constructs the QR decomposition of a matrix with elements of type double.

a A double matrix to be factored.

Methods

GetPermute

public int[] GetPermute()

Returns an int array containing information about the permutation of the elements of the matrix during pivoting.

Returns — The k-th element contains the index of the column of the matrix that has been interchanged into the k-th column.

GetQ

public double[,] GetQ()

The orthogonal or unitary matrix Q.

Returns — A double matrix containing the accumulated orthogonal matrix Q from the QR decomposition.

GetR

public double[,] GetR()

The upper trapezoidal matrix R.

Returns — The upper trapezoidal double matrix R of the QR decomposition.

GetRank

public int GetRank()

Returns the rank of the matrix used to construct this instance.

Returns — An int specifying the rank of the matrix used to construct this instance.

GetRank

public int GetRank(double tolerance)

Returns the rank of the matrix given an input tolerance.
tolerance  A double scalar value used in determining the rank of the matrix.

Returns — An int specifying the rank of the matrix.

Solve

public double[] Solve(double[] b)
Returns the solution to the least-squares problem Ax = b.

b  A double array to be manipulated.

Returns — A double array containing the solution vector to Ax = b with components corresponding to the unused columns set to zero.

Imsl.Math.SingularMatrixException is thrown when the upper triangular matrix R resulting from the QR factorization is singular

Solve

public double[] Solve(double[] b, double tol)
Returns the solution to the least-squares problem Ax = b using an input tolerance.

b  A double array to be manipulated.

tol  A double scalar value used in determining the rank of A.

Returns — A double array containing the solution vector to Ax = b with components corresponding to the unused columns set to zero.

Imsl.Math.SingularMatrixException is thrown when the upper triangular matrix R resulting from the QR factorization is singular

Description

Class QR computes the QR decomposition of a matrix using Householder transformations. It is based on the LINPACK routine SQRDC; see Dongarra et al. (1979).

QR determines an orthogonal matrix Q, a permutation matrix P, and an upper trapezoidal matrix R with diagonal elements of nonincreasing magnitude, such that AP = QR. The Householder transformation for column $k$ is of the form

$$I - \frac{uu^T}{P_k}$$

for $k = 1, 2, \ldots, \min(\text{number of rows of } A, \text{number of columns of } A)$, where $u$ has zeros in the first $k - 1$ positions. The matrix $Q$ is not produced directly by QR. Instead the information needed to reconstruct the Householder transformations is saved. If the matrix $Q$ is needed explicitly, use the $Q$ property. This method accumulates $Q$ from its factored form.

Before the decomposition is computed, initial columns are moved to the beginning of the array A and the final columns to the end. Both initial and final columns are frozen in place during the computation. Only free columns are pivoted. Pivoting is done on the free columns of largest reduced norm.
Example: QR Factorization of a Matrix

The QR Factorization of a Matrix is performed. A linear system is then solved using the factorization. The rank of the input matrix is also computed.

```csharp
using System;
using Imsl.Math;

public class QREx1
{
    public static void Main(String[] args)
    {
        double[,] a = {
            {1, 2, 4},
            {1, 4, 16},
            {1, 6, 36},
            {1, 8, 64}
        };
        double[] b = new double[]{16.99, 57.01, 120.99, 209.01};

        // Compute the QR factorization of A
        QR qr = new QR(a);

        // Solve Ax = b
        double[] x = qr.Solve(b);
        new PrintMatrix("x").Print(x);

        // Print Q and R.
        new PrintMatrix("Q").Print(qr.GetQ());
        new PrintMatrix("R").Print(qr.GetR());

        // Find the rank of A.
        int rank = qr.GetRank();
        Console.Out.WriteLine("rank = " + rank);
    }
}
```

Output

```
x
0
0 0.990000000000019
1 2.00199999999999
2 3

Q
0 -0.0531494003452735 -0.54217094609664 0.808223859120487 -0.22360679774998
1 -0.212597601381094 -0.657435635424271 -0.269407953040162 0.670820393249937
2 -0.478344603107461 -0.345794067982896 -0.449013255066938 -0.670820393249936
3 -0.850390405524374 0.392753756227487 0.269407953040163 0.223606797749979
```
SVD Class

Singular Value Decomposition (SVD) of a rectangular matrix of type double.

public class Imsl.Math.SVD

Properties

Info

public int Info {get; }
Returns the index of the first singular value for which the algorithm converged. Convergence was obtained for the Info, Info+1, ..., Min(nrows,ncols) singular values and their corresponding vectors. Here, nrows and ncols represent the number of rows and columns of the input matrix respectively.

Rank

public int Rank {get; }
Returns the rank of the matrix used to construct this instance. An int scalar containing the rank of the matrix used to construct this instance. The estimated rank of the input matrix is the number of singular values which are larger than a tolerance.

Constructors

SVD

public SVD(double[,] a, double tol)
Construct the singular value decomposition of a rectangular matrix with a given tolerance.

If tol is positive, then a singular value is considered negligible if the singular value is less than or equal to tol. If tol is negative, then a singular value is considered negligible if the singular value is less than or equal to the absolute value of the product of tol and the infinity norm of the input matrix. In the latter case, the absolute value of tol generally contains an estimate of the level of the relative error in the data.

a A double matrix for which the singular value decomposition is to be computed.
tol A double scalar containing the tolerance used to determine when a singular value is negligible.

Imsl.Math.DidNotConvergeException is thrown when the rank cannot be determined because convergence was not obtained for all singular values

SVD

public SVD(double[,] a)
Construct the singular value decomposition of a rectangular matrix with default tolerance.

The tolerance used is 2.2204460492503e-14. This tolerance is used to determine rank. A singular value is considered negligible if the singular value is less than or equal to this tolerance.

a A double matrix for which the singular value decomposition is to be computed.

Methods

GetS

public double[] GetS()
Returns the singular values.

Returns — A double array containing the singular values of the matrix.

GetU

public double[,] GetU()
Returns the left singular vectors.

Returns — A double matrix containing the left singular vectors.

GetV

public double[,] GetV()
Returns the right singular vectors.

Returns — A double matrix containing the right singular vectors

Inverse

public double[,] Inverse()
Compute the Moore-Penrose generalized inverse of a real matrix.

Returns — A double matrix containing the generalized inverse of the matrix used to construct this instance.

Description

SVD is based on the LINPACK routine SSVDC; see Dongarra et al. (1979).

Let n be the number of rows in A and let p be the number of columns in A. For any
For an \( n \times p \) matrix \( A \), there exists an \( n \times n \) orthogonal matrix \( U \) and a \( p \times p \) orthogonal matrix \( V \) such that

\[
U^T AV = \begin{cases} 
\begin{bmatrix} \Sigma & 0 \\ 0 & 0 \end{bmatrix} & \text{if } n \geq p \\
\begin{bmatrix} \Sigma & 0 \\ 0 & 0 \end{bmatrix} & \text{if } n \leq p 
\end{cases}
\]

where \( \Sigma = \text{diag}(\sigma_1, \ldots, \sigma_m) \), and \( m = \min(n, p) \). The scalars \( \sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_m \geq 0 \) are called the singular values of \( A \). The columns of \( U \) are called the left singular vectors of \( A \). The columns of \( V \) are called the right singular vectors of \( A \).

The estimated rank of \( A \) is the number of \( \sigma_k \) that is larger than a tolerance \( \eta \). If \( \tau \) is the parameter \texttt{tol} in the program, then

\[
\eta = \begin{cases} 
\tau & \text{if } \tau > 0 \\
|\tau| \|A\|_\infty & \text{if } \tau < 0 
\end{cases}
\]

The Moore-Penrose generalized inverse of the matrix is computed by partitioning the matrices \( U, V \) and \( \Sigma \) as \( U = (U_1, U_2), V = (V_1, V_2) \) and \( \Sigma_1 = \text{diag}(\sigma_1, \ldots, \sigma_k) \) where the "1" matrices are \( k \) by \( k \). The Moore-Penrose generalized inverse is \( V_1 \Sigma_1^{-1} U_1^T \).

**Example: Singular Value Decomposition of a Matrix**

The singular value decomposition of a matrix is performed. The rank of the matrix is also computed.

```csharp
using System;
using Imsl.Math;

public class SVDEx1
{
    public static void Main(String[] args)
    {
        double[,] a = {
            {1, 2, 1, 4},
            {3, 2, 1, 3},
            {4, 3, 1, 4},
            {2, 1, 3, 1},
            {1, 5, 2, 2},
            {1, 2, 2, 3}
        };

        // Compute the SVD factorization of A
        SVD svd = new SVD(a);

        // Print U, S and V.
        new PrintMatrix("U").SetPageWidth(80).Print(svd.GetU());
        new PrintMatrix("S").SetPageWidth(80).Print(svd.GetS());
    }
}
```

**Linear Systems**

**SVD Class**

- 25
new PrintMatrix("V").SetPageWidth(80).Print(svd.GetV());

// Find the rank of A.
int rank = svd.Rank;
Console.Out.WriteLine("rank = " + rank);
}
}

Output

<table>
<thead>
<tr>
<th></th>
<th>U</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.119670992640587</td>
<td>0.439082824383239</td>
</tr>
<tr>
<td>1</td>
<td>0.403753713172442</td>
<td>0.345110837105607</td>
<td>0.0565761852901658</td>
</tr>
<tr>
<td>2</td>
<td>0.545120486248343</td>
<td>0.429264893493195</td>
<td>0.0513926928086694</td>
</tr>
<tr>
<td>3</td>
<td>0.264784294004146</td>
<td>0.0683195253271513</td>
<td>0.883860867430429</td>
</tr>
<tr>
<td>4</td>
<td>0.446310112301556</td>
<td>0.816827623278282</td>
<td>0.141899675060401</td>
</tr>
<tr>
<td>5</td>
<td>0.354628656614145</td>
<td>-0.102147399162125</td>
<td>-0.0043844397986985</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>V</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-0.565399858908374</td>
<td>0.0243115161463761</td>
</tr>
<tr>
<td>1</td>
<td>0.214775757622681</td>
<td>0.80890058872827</td>
<td>0.11929741721493</td>
</tr>
<tr>
<td>2</td>
<td>0.432144162809737</td>
<td>-0.572327648171096</td>
<td>0.0403309248707933</td>
</tr>
<tr>
<td>3</td>
<td>-0.215253681829747</td>
<td>-0.062509228900579</td>
<td>-0.3062166996707105</td>
</tr>
<tr>
<td>4</td>
<td>0.321268854269887</td>
<td>0.0621337820958098</td>
<td>-0.0799352679995222</td>
</tr>
<tr>
<td>5</td>
<td>-0.5459800221853269</td>
<td>-0.0987946265624981</td>
<td>0.746739576113111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11.4850179115597</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3.2697512144125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.65336516200783</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.08872967244092</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>V</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-0.444294128842354</td>
<td>0.555531257799947</td>
</tr>
<tr>
<td>1</td>
<td>-0.558067238190387</td>
<td>-0.654298740112323</td>
<td>0.277466900458814</td>
</tr>
<tr>
<td>2</td>
<td>-0.324386103206282</td>
<td>-0.351360645592613</td>
<td>-0.732099653429598</td>
</tr>
<tr>
<td>3</td>
<td>-0.621238553843379</td>
<td>0.373930310383434</td>
<td>0.444401954223745</td>
</tr>
</tbody>
</table>

rank = 4

26 • SVD Class

IMSL C# Numerical Library
Chapter 2: Eigensystem Analysis

Types

class Eigen ................................................................. 29
class SymEigen ............................................................ 31

Usage Notes

An ordinary linear eigensystem problem is represented by the equation $Ax = \lambda x$ where $A$ denotes an $n \times n$ matrix. The value $\lambda$ is an eigenvalue and $x \neq 0$ is the corresponding eigenvector. The eigenvector is determined up to a scalar factor. In all functions, we have chosen this factor so that $x$ has Euclidean length one, and the component of $x$ of largest magnitude is positive. If $x$ is a complex vector, this component of largest magnitude is scaled to be real and positive. The entry where this component occurs can be arbitrary for eigenvectors having nonunique maximum magnitude values.

Error Analysis and Accuracy

Except in special cases, functions will not return the exact eigenvalue-eigenvector pair for the ordinary eigenvalue problem $Ax = \lambda x$. Typically, the computed pair

$$\tilde{x}, \tilde{\lambda}$$

is an exact eigenvector-eigenvalue pair for a "nearby" matrix $A + E$. Information about $E$ is known only in terms of bounds of the form $\|E\|_2 \leq f(n) \|A\|_2 \varepsilon$. The value of $f(n)$ depends on the algorithm, but is typically a small fractional power of $n$. The parameter $\varepsilon$ is the machine precision. By a theorem due to Bauer and Fike (see Golub and Van Loan 1989, p. 342),

$$\min |\tilde{\lambda} - \lambda| \leq \kappa(X) \|E\|_2$$

for all $\lambda \in \sigma(A)$

where $\sigma(A)$ is the set of all eigenvalues of $A$ (called the spectrum of $A$), $X$ is the matrix of
eigenvectors, $\|\cdot\|_2$ is Euclidean length, and $\kappa(X)$ is the condition number of $X$ defined as $\kappa(X) = \|X\|_2 \|X^{-1}\|_2$. If $A$ is a real symmetric or complex Hermitian matrix, then its eigenvector matrix $X$ is respectively orthogonal or unitary. For these matrices, $\kappa(X) = 1$.

The accuracy of the computed eigenvalues

$$\hat{\lambda}_j$$

and eigenvectors

$$\hat{x}_j$$

can be checked by computing their performance index $\tau$. The performance index is defined to be

$$\tau = \max_{1 \leq j \leq n} \frac{\|A\hat{x}_j - \hat{\lambda}_j\hat{x}_j\|_2}{n\varepsilon \|A\|_2 \|\hat{x}_j\|_2}$$

where $\varepsilon$ is again the machine precision.

The performance index $\tau$ is related to the error analysis because

$$\|E\hat{x}_j\|_2 = \left\|A\hat{x}_j - \hat{\lambda}_j\hat{x}_j\right\|_2$$

where $E$ is the "nearby" matrix discussed above.

While the exact value of $\tau$ is precision and data dependent, the performance of an eigensystem analysis function is defined as excellent if $\tau < 1$, good if $1 \leq \tau \leq 100$, and poor if $\tau > 100$. This is an arbitrary definition, but large values of $\tau$ can serve as a warning that there is a significant error in the calculation.

If the condition number $\kappa(X)$ of the eigenvector matrix $X$ is large, there can be large errors in the eigenvalues even if $\tau$ is small. In particular, it is often difficult to recognize near multiple eigenvalues or unstable mathematical problems from numerical results. This facet of the eigenvalue problem is often difficult for users to understand. Suppose the accuracy of an individual eigenvalue is desired. This can be answered approximately by computing the *condition number of an individual eigenvalue*(see Golub and Van Loan 1989, pp. 344-345). For matrices $A$, such that the computed array of normalized eigenvectors $X$ is invertible, the condition number of $\lambda_i$ is

$$\kappa_j = \|e_j^T X^{-1}\|,$$

the Euclidean length of the $j$-th row of $X^{-1}$. Users can choose to compute this matrix using the class LU in "Linear Systems." An approximate bound for the accuracy of a computed eigenvalue is then given by $\kappa_j\varepsilon \|A\|$. To compute an approximate bound for the relative accuracy of an eigenvalue, divide this bound by $|\lambda_j|$.
Eigen Class

Collection of Eigen System functions.

public class Imsl.Math.Eigen

Constructors

Eigen

public Eigen(double[,] a)
Constructs the eigenvalues and the eigenvectors of a real square matrix.

a A double square matrix whose eigensystem is to be constructed.

Imsl.Math.DidNotConvergeException is thrown when the algorithm fails to converge on the eigenvalues of the matrix

Eigen

public Eigen(double[,] a, bool computeVectors)
Constructs the eigenvalues and (optionally) the eigenvectors of a real square matrix.

a A double square matrix whose eigensystem is to be constructed.
computeVectors A bool value of true if the eigenvectors are to be computed.

Imsl.Math.DidNotConvergeException is thrown when the algorithm fails to converge on the eigenvalues of the matrix

Methods

GetValues

public Imsl.Math.Complex[] GetValues()
Returns the eigenvalues of a matrix of type double.

Returns — A Complex array containing the eigenvalues of this matrix in descending order.

GetVectors

public Imsl.Math.Complex[,] GetVectors()
Returns the eigenvectors.

Returns — A Complex matrix containing the eigenvectors. The eigenvector corresponding to the j-th eigenvalue is stored in the j-th column. Each vector is normalized to have Euclidean length one.
PerformanceIndex
    public double PerformanceIndex(double[,] a)
Returns the performance index of a real eigensystem.

    A performance index less than 1 is considered excellent, 1 to 100 is good, while greater
    than 100 is considered poor.

    a  A double matrix.

Returns — A double scalar value indicating how well the algorithms which have
computed the eigenvalue and eigenvector pairs have performed.

Description

Eigen computes the eigenvalues and eigenvectors of a real matrix. The matrix is first balanced.
Orthogonal similarity transformations are used to reduce the balanced matrix to a real upper
Hessenberg matrix. The implicit double-shifted QR algorithm is used to compute the
eigenvalues and eigenvectors of this Hessenberg matrix. The eigenvectors are normalized such
that each has Euclidean length of value one. The largest component is real and positive.

The balancing routine is based on the EISPACK routine BALANC. The reduction routine is
based on the EISPACK routines ORTHES and ORTRAN. The QR algorithm routine is based on the
EISPACK routine HQR2. See Smith et al. (1976) for the EISPACK routines. Further details,
some timing data, and credits are given in Hanson et al. (1990).

While the exact value of the performance index, $\tau$, is highly machine dependent, the
performance of Eigen is considered excellent if $\tau < 1$, good if $1 \leq \tau \leq 100$, and poor if $\tau > 100$.

The performance index was first developed by the EISPACK project at Argonne National
Laboratory; see Smith et al. (1976, pages 124-125).

Example: Eigensystem Analysis

The eigenvalues and eigenvectors of a matrix are computed.

using System;
using Imsl.Math;

public class EigenEx1
{
    public static void Main(String[] args)
    {
        double[,] a = {
            {8, -1, -5},
            {-4, 4, -2},
            {18, -5, -7}
        };
        Eigen eigen = new Eigen(a);
        new PrintMatrix("Eigenvalues").SetPageWidth(80).Print(eigen.GetValues());
        new PrintMatrix("Eigenvectors").SetPageWidth(80).Print(eigen.GetVectors());
    }
}
Output

Eigenvalues

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2+4i</td>
<td>2-4i</td>
</tr>
<tr>
<td>2</td>
<td>0.999999999999997</td>
<td></td>
</tr>
</tbody>
</table>

Eigenvectors

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.316227766016838-0.316227766016838i</td>
<td>0.632455532033676</td>
</tr>
<tr>
<td>2</td>
<td>1.6653453693773E-16-0.632455532033676i</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.316227766016838+0.316227766016838i</td>
<td>0.632455532033676</td>
</tr>
<tr>
<td>2</td>
<td>1.6653453693773E-16+0.632455532033676i</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.408248290463863</td>
<td>0.816496580927725</td>
</tr>
<tr>
<td>2</td>
<td>0.408248290463864</td>
<td></td>
</tr>
</tbody>
</table>

SymEigen Class

Computes the eigenvalues and eigenvectors of a real symmetric matrix.

```java
public class Imsl.Math.SymEigen
```

Constructors

SymEigen

```java
    public SymEigen(double[,] a)
    Constructs the eigenvalues and the eigenvectors for a real symmetric matrix.

    a    The symmetric matrix whose eigensystem is to be constructed.
```

SymEigen

```java
    public SymEigen(double[,] a, bool computeVectors)
    Constructs the eigenvalues and (optionally) the eigenvectors for a real symmetric matrix.
```
A double symmetric matrix whose eigensystem is to be constructed.

computeVectors A boolean, true if the eigenvectors are to be computed.

Methods

GetValues

public double[] GetValues()
Returns the eigenvalues.
If the algorithm fails to converge on an eigenvalue, that eigenvalue is set to NaN.
Returns — A double array containing the eigenvalues in descending order.

GetVectors

public double[,] GetVectors()
Return the eigenvectors of a symmetric matrix of type double.
The j-th column of the eigenvector matrix corresponds to the j-th eigenvalue. The
eigenvectors are normalized to have Euclidean length one. If the eigenvectors were not
computed by the constructor, then null is returned.
Returns — A double array containing the eigenvectors.

PerformanceIndex

public double PerformanceIndex(double[,] a)
Returns the performance index of a real symmetric eigensystem.
A performance index less than 1 is considered excellent, 1 to 100 is good, while greater
than 100 is considered poor.
a A double symmetric matrix.
Returns — A double scalar value indicating how well the algorithms which have
computed the eigenvalue and eigenvector pairs have performed.

Description

Orthogonal similarity transformations are used to reduce the matrix to an equivalent symmetric
tridiagonal matrix. These transformations are accumulated. An implicit rational QR algorithm
is used to compute the eigenvalues of this tridiagonal matrix. The eigenvectors are computed
using the eigenvalues as perfect shifts, Parlett (1980, pages 169, 172). The reduction routine is
based on the EISPACK routine TRED2. See Smith et al. (1976) for the EISPACK routines.
Further details, some timing data, and credits are given in Hanson et al. (1990).

Let $M$ = the number of eigenvalues, $\lambda$ = the array of eigenvalues, and $x_j$ is the associated
eigenvector with jth eigenvalue.

Also, let $\varepsilon$ be the machine precision. The performance index, $\tau$, is defined to be

$$
\tau = \max_{1 \leq j \leq M} \frac{\|Ax_j - \lambda_jx_j\|}{10N\varepsilon \|A\|_1 \|x_j\|_1}
$$
While the exact value of τ is highly machine dependent, the performance of SymEigen is considered excellent if τ < 1, good if 1 ≤ 100, and poor if τ > 100. The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Smith et al. (1976, pages 124-125).

**Example: Eigenvalues and Eigenvectors of a Symmetric Matrix**

The eigenvalues and eigenvectors of a symmetric matrix are computed.

```csharp
using System;
using Imsl.Math;

public class SymEigenEx1
{
    public static void Main(String[] args)
    {
        double[,] a = {
            {1, 1, 1},
            {1, 1, 1},
            {1, 1, 1}
        };

        SymEigen eigen = new SymEigen(a);
        new PrintMatrix("Eigenvalues").Print(eigen.GetValues());
        new PrintMatrix("Eigenvectors").Print(eigen.GetVectors());
    }
}
```

**Output**

**Eigenvalues**

0 3
0 0 3
1 -3.62597321469472E-16
2 -2.22044604925031E-16

**Eigenvectors**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.577350269189626</td>
<td>0.816496580927726</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.577350269189626</td>
<td>-0.408248290463863</td>
<td>-0.707106781186547</td>
</tr>
<tr>
<td>2</td>
<td>0.577350269189626</td>
<td>-0.408248290463863</td>
<td>0.707106781186548</td>
</tr>
</tbody>
</table>
Chapter 3: Interpolation and Approximation

Types

- `class Spline` ................................................................. 37
- `class CsAkima` ............................................................. 39
- `class CsInterpolate` ...................................................... 41
- `enumeration CsInterpolate.Condition` .............................. 44
- `class CsPeriodic` .......................................................... 45
- `class CsShape` ............................................................. 47
- `class CsSmooth` ............................................................ 50
- `class CsSmoothC2` ......................................................... 53
- `class BSpline` ............................................................... 57
- `class BsInterpolate` ....................................................... 60
- `class BsLeastSquares` .................................................... 63
- `class RadialBasis` .......................................................... 66
- `interface RadialBasis.IFunction` .................................... 71
- `class RadialBasis.Gaussian` ............................................ 71
- `class RadialBasis.HardyMultiquadric` ............................... 72

Usage Notes

This chapter contains classes to interpolate and approximate data with cubic splines. Interpolation means that the fitted curve passes through all of the specified data points. An approximating spline does not have to pass through any of the data points. An approximating curve can therefore be smoother than an interpolating curve.

Cubic splines are smooth $C^1$ or $C^2$ fourth-order piecewise-polynomial (pp) functions. For historical and other reasons, cubic splines are the most heavily used pp functions.

This chapter contains four cubic spline interpolation classes and two approximation classes. These classes are derived from the base class `Spline`, which provides basic services, such as
spline evaluation and integration.

The chart shows how the six cubic splines in this chapter fit a single data set.

Class **CsInterpolate** allows the user to specify various endpoint conditions (such as the value of the first and second derivatives at the right and left endpoints).

Class **CsPeriodic** is used to fit periodic (repeating) data. The sample data set used is not periodic and so the curve does not pass through the final data point.

Class **CsAkima** keeps the shape of the data while minimizing oscillations.

Class **CsShape** keeps the shape of the data by preserving its convexity.

Class **CsSmooth** constructs a smooth spline from noisy data.

Class **CsSmoothC2** constructs a smooth spline from noisy data using cross-validation and a user-supplied smoothing parameter.
**Spline Class**

*Spline* represents and evaluates univariate piecewise polynomial splines.

```java
public class Imsl.Math.Spline
```

**Constructor**

Spline

```java
Spline()
```
Initializes a new instance of the Imsl.Math.Spline (p. 37) class.

**Methods**

**Derivative**

```java
virtual public double[] Derivative(double[] x, int ideriv)
```
Returns the value of the derivative of the spline at each point of an array.

- `x` A `double` array of points at which the derivative is to be evaluated.
- `ideriv` An `int` specifying the derivative to be computed. If zero, the function value is returned. If one, the first derivative is returned, etc.

Returns — A `double` array containing the value of the derivative the spline at each point of the array `x`.

**Derivative**

```java
virtual public double Derivative(double x, int ideriv)
```
Returns the value of the derivative of the spline at a point.

- `x` A `double`, the point at which the derivative is to be evaluated.
- `ideriv` An `int` specifying the derivative to be computed. If zero, the function value is returned. If one, the first derivative is returned, etc.

Returns — A `double` containing the value of the derivative of the spline at the point `x`.

**Derivative**

```java
virtual public double Derivative(double x)
```
Returns the value of the first derivative of the spline at a point.

- `x` A `double`, the point at which the derivative is to be evaluated.

Returns — A `double` containing the value of the first derivative of the spline at the point `x`.
Eval

\[ \text{virtual public double[]} \quad \text{Eval(double[]} \times \text{)} \]

Returns the value of the spline at each point of an array.

\[ \times \quad \text{A double array of points at which the spline is to be evaluated.} \]

Returns — A double array containing the value of the spline at each point of the array \( x \).

Eval

\[ \text{virtual public double} \quad \text{Eval(double} \times \text{)} \]

Returns the value of the spline at a point.

\[ \times \quad \text{A double, the point at which the spline is to be evaluated.} \]

Returns — A double giving the value of the spline at the point \( x \).

GetBreakpoints

\[ \text{public double[]} \quad \text{GetBreakpoints()} \]

Returns a copy of the breakpoints.

Returns — A double array containing a copy of the breakpoints.

Integral

\[ \text{virtual public double} \quad \text{Integral(double} \times, \text{double} \times \text{)} \]

Returns the value of an integral of the spline.

\[ \times \quad \text{A double specifying the lower limit of integration.} \]
\[ \times \quad \text{A double specifying the upper limit of integration.} \]

Returns — A double, the integral of the spline from \( a \) to \( b \).

Description

A univariate piecewise polynomial (function) \( p(x) \) is specified by giving its breakpoint sequence \( \text{breakPoint[]} = \xi \in \mathbb{R}^n \), the order \( k \) (degree \( k-1 \)) of its polynomial pieces, and the \( k \times (n-1) \) matrix \( \text{coef} = c \) of its local polynomial coefficients. In terms of this information, the piecewise polynomial (ppoly) function is given by

\[
p(x) = \sum_{j=1}^{k} c_{ji} \frac{(x - \xi_i)^{j-1}}{(j-1)!} \quad \text{for } \xi_i \leq x \leq \xi_{i+1}
\]

The breakpoint sequence \( \xi \) is assumed to be strictly increasing, and we extend the ppoly function to the entire real axis by extrapolation from the first and last intervals.
CsAkima Class

Extension of the Spline class to handle the Akima cubic spline.

public class Imsl.Math.CsAkima : Spline

Constructor

CsAkima

public CsAkima(double[] xData, double[] yData)
Constructs the Akima cubic spline interpolant to the given data points.

xData  A double array containing the x-coordinates of the data. Values must be distinct.

yData  A double array containing the y-coordinates of the data.

System.ArgumentException is thrown if the arrays xData and yData do not have the same length

Methods

Derivative

virtual public double[] Derivative(double[] x, int ideriv)
Returns the value of the derivative of the spline at each point of an array.

Derivative

virtual public double Derivative(double x, int ideriv)
Returns the value of the derivative of the spline at a point.

Derivative

virtual public double Derivative(double x)
Returns the value of the first derivative of the spline at a point.

Eval

virtual public double[] Eval(double[] x)
Returns the value of the spline at each point of an array.
Eval

```
virtual public double Eval(double x)
Returns the value of the spline at a point.
```

GetBreakpoints

```
public double[] GetBreakpoints()
Returns a copy of the breakpoints.
```

Integral

```
virtual public double Integral(double a, double b)
Returns the value of an integral of the spline.
```

**Description**

Class **CsAkima** computes a $C^1$ cubic spline interpolant to a set of data points $(x_i, f_i)$ for $i = 0, \ldots, n - 1$. The breakpoints of the spline are the abscissas. Endpoint conditions are automatically determined by the program; see Akima (1970) or de Boor (1978).

If the data points arise from the values of a smooth, say $C^4$, function $f$, i.e. $f_i = f(x_i)$, then the error will behave in a predictable fashion. Let $\xi$ be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$
\| f - s \|_{[\xi_0, \xi_{n-1}]} \leq C \left\| f^{(2)} \right\|_{[\xi_0, \xi_{n-1}]} |\xi|^2
$$

where

$$
|\xi| := \max_{i=1, \ldots, n-1} |\xi_i - \xi_{i-1}|
$$

**CsAkima** is based on a method by Akima (1970) to combat wiggles in the interpolant. The method is nonlinear; and although the interpolant is a piecewise cubic, cubic polynomials are not reproduced. (However, linear polynomials are reproduced.)

**Example: The Akima cubic spline interpolant**

A cubic spline interpolant to a function is computed. The value of the spline at point 0.25 is printed.
using System;
using Imsl.Math;

public class CsAkimaEx1
{
    public static void Main(String[] args)
    {
        int n = 11;
        double[] x = new double[n];
        double[] y = new double[n];

        for (int k = 0; k < n; k++)
        {
            x[k] = (double) k / (double) (n - 1);
            y[k] = Math.Sin(15.0 * x[k]);
        }

        CsAkima cs = new CsAkima(x, y);
        double csv = cs.Eval(0.25);
        Console.Out.WriteLine("The computed cubic spline value at " +
        "point .25 is " + csv);
    }
}

Output

The computed cubic spline value at point .25 is -0.478185519991867

---

**CsInterpolate Class**

Extension of the Spline class to interpolate data points.

**public class Imsl.Math.CsInterpolate : Spline**

**Constructors**

CsInterpolate

**public CsInterpolate(double[] xData, double[] yData)**

Constructs a cubic spline that interpolates the given data points.

- **xData**  A double array containing the x-coordinates of the data. Values must be distinct.
- **yData**  A double array containing the y-coordinates of the data. The arrays **xData** and **yData** must have the same length.
CsInterpolate
public CsInterpolate(double[] xData, double[] yData,
    Imsl.Math(CsInterpolate.Condition typeLeft, double valueLeft,
    Imsl.Math(CsInterpolate.Condition typeRight, double valueRight)
) Constructs a cubic spline that interpolates the given data points with specified derivative endpoint conditions.

xData  A double array containing the x-coordinates of the data. Values must be distinct.
yData  A double array containing the y-coordinates of the data. The arrays xData and yData must have the same length.
typeLeft  A CsInterpolate.Condition denoting the type of condition at the left endpoint. This can be NotAKnot, FirstDerivative or SecondDerivative.
valueLeft  A double value at the left endpoint. If typeLeft is NotAKnot this is ignored, Otherwise, it is the value of the specified derivative.
typeRight  A CsInterpolate.Condition denoting the type of condition at the right endpoint. This can be NotAKnot, FirstDerivative or SecondDerivative.
valueRight  A double value at the right endpoint.

Methods

Derivative
virtual public double[] Derivative(double[] x, int ideriv)
Returns the value of the derivative of the spline at each point of an array.

Derivative
virtual public double Derivative(double x, int ideriv)
Returns the value of the derivative of the spline at a point.

Derivative
virtual public double Derivative(double x)
Returns the value of the first derivative of the spline at a point.

Eval
virtual public double[] Eval(double[] x)
Returns the value of the spline at each point of an array.
Eval
    virtual public double Eval(double x)
    Returns the value of the spline at a point.

GetBreakpoints
    public double[] GetBreakpoints()
    Returns a copy of the breakpoints.

Integral
    virtual public double Integral(double a, double b)
    Returns the value of an integral of the spline.

Description

CsInterpolate computes a $C^2$ cubic spline interpolant to a set of data points $(x_i, f_i)$ for $i = 0, ..., n-1$. The breakpoints of the spline are the abscissas. Endpoint conditions are automatically determined by the program. These conditions correspond to the "not-a-knot" condition (see de Boor 1978), which requires that the third derivative of the spline be continuous at the second and next-to-last breakpoint. If $n$ is 2 or 3, then the linear or quadratic interpolating polynomial is computed, respectively.

If the data points arise from the values of a smooth, say, $C^4$ function $f$, i.e. $f_i = f(x_i)$, then the error will behave in a predictable fashion. Let $\xi$ be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$|f - s|_{[\xi_0,\xi_n]} \leq C \left\| f^{(4)} \right\|_{[\xi_0,\xi_n]} |\xi|^4$$

where

$$|\xi| := \max_{i=0,\ldots,n-1} |\xi_{i+1} - \xi_i|$$

For more details, see de Boor (1978, pages 55-56).

Example: The cubic spline interpolant

A cubic spline interpolant to a function is computed. The value of the spline at point 0.25 is printed.
using System;
using Imsl.Math;

class CsInterpolateEx1
{
    public static void Main(String[] args)
    {
        int n = 11;
        double[] x = new double[n];
        double[] y = new double[n];

        for (int k = 0; k < n; k++)
        {
            x[k] = (double) k / (double) (n - 1);
            y[k] = System.Math.Sin(15.0 * x[k]);
        }

        CsInterpolate cs = new CsInterpolate(x, y);
        double csv = cs.Eval(0.25);
        Console.Out.WriteLine("The computed cubic spline value at point .25 is " + " + csv);    
    }
}

Output
The computed cubic spline value at point .25 is -0.548772503812158

---

CsInterpolate.Condition Enumeration

Denotes the type of condition at an endpoint.

public enumeration Imsl.Math.CsInterpolate.Condition

Fields

FirstDerivative
    public Imsl.Math.CsInterpolate.Condition FirstDerivative
    Satisfies the endpoint condition of the first derivative at the right and left points.

NotAKnot
    public Imsl.Math.CsInterpolate.Condition NotAKnot
    Satisfies the "not-a-knot" condition.

SecondDerivative

---
Satisfies the endpoint condition of the second derivative at the right and left points.

---

**CsPeriodic Class**

Extension of the Spline class to interpolate data points with periodic boundary conditions.

```csharp
public class Imsl.Math.CsPeriodic : Spline

**Constructor**

`CsPeriodic(double[] xData, double[] yData)`
Constructs a cubic spline that interpolates the given data points with periodic boundary conditions.

- **xData** A `double` array containing the x-coordinates of the data. There must be at least 4 data points and values must be distinct.
- **yData** A `double` array containing the y-coordinates of the data. The arrays `xData` and `yData` must have the same length.

**Methods**

`Derivative(double[] x, int ideriv)`
Returns the value of the derivative of the spline at each point of an array.

`Derivative(double x, int ideriv)`
Returns the value of the first derivative of the spline at a point.

`Derivative(double x)`
Returns the value of the first derivative of the spline at a point.

---
Eval

    virtual public double[] Eval(double[] x)
    Returns the value of the spline at each point of an array.

Eval

    virtual public double Eval(double x)
    Returns the value of the spline at a point.

GetBreakpoints

    public double[] GetBreakpoints()
    Returns a copy of the breakpoints.

Integral

    virtual public double Integral(double a, double b)
    Returns the value of an integral of the spline.

Description

Class CsPeriodic computes a $C^2$ cubic spline interpolant to a set of data points $(x_i, f_i)$ for $i = 0, \ldots, n - 1$. The breakpoints of the spline are the abscissas. The program enforces periodic endpoint conditions. This means that the spline $s$ satisfies $s(a) = s(b)$, $s'(a) = s'(b)$, and $s''(a) = s''(b)$, where $a$ is the leftmost abscissa and $b$ is the rightmost abscissa. If the ordinate values corresponding to $a$ and $b$ are not equal, then a warning message is issued. The ordinate value at $b$ is set equal to the ordinate value at $a$ and the interpolant is computed.

If the data points arise from the values of a smooth (say $C^4$) periodic function $f$, i.e. $f_i = f(x_i)$, then the error will behave in a predictable fashion. Let $\xi$ be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$|f - s|_{[\xi_0, \xi_{n-1}]} \leq C|f^{(4)}|_{[\xi_0, \xi_{n-1}]} |\xi|^{4}$$

where

$$|\xi| := \max_{i=1,\ldots,n-1} |\xi_i - \xi_{i-1}|$$

For more details, see de Boor (1978, pages 320-322).
Example: The cubic spline interpolant with periodic boundary conditions

A cubic spline interpolant to a function is computed. The value of the spline at point 0.23 is printed.

```csharp
using System;
using Imsl.Math;

public class CsPeriodicEx1
{
    public static void Main(String[] args)
    {
        int n = 11;
        double[] x = new double[n];
        double[] y = new double[n];

        double h = 2.0 * System.Math.PI / 15.0 / 10.0;
        for (int k = 0; k < n; k++)
        {
            x[k] = h * (double) (k);
            y[k] = System.Math.Sin(15.0 * x[k]);
        }

        CsPeriodic cs = new CsPeriodic(x, y);
        double csv = cs.Eval(0.23);
        Console.Out.WriteLine("The computed cubic spline value at " + "point .23 is " + csv);
    }
}
```

Output

The computed cubic spline value at point .23 is -0.303401472606451

---

**CsShape Class**

Extension of the Spline class to interpolate data points consistent with the concavity of the data.

```csharp
public class Imsl.Math.CsShape : Spline

Constructor

CsShape
   public CsShape(double[] xData, double[] yData)
```

---

Interpolation and Approximation
Construct a cubic spline interpolant which is consistent with the concavity of the data.

**xData**  A double array containing the x-coordinates of the data. Values must be distinct.

**yData**  A double array containing the y-coordinates of the data. The arrays xData and yData must have the same length.

**Imsl.Math.TooManyIterationsException** is thrown if the iteration did not converge.
**Imsl.Math.SingularMatrixException** is thrown if matrix is singular.

### Methods

**Derivative**

```csharp
virtual public double[] Derivative(double[] x, int ideriv)
Returns the value of the derivative of the spline at each point of an array.
```

**Derivative**

```csharp
virtual public double Derivative(double x, int ideriv)
Returns the value of the derivative of the spline at a point.
```

**Derivative**

```csharp
virtual public double Derivative(double x)
Returns the value of the first derivative of the spline at a point.
```

**Eval**

```csharp
virtual public double[] Eval(double[] x)
Returns the value of the spline at each point of an array.
```

**Eval**

```csharp
virtual public double Eval(double x)
Returns the value of the spline at a point.
```

**GetBreakpoints**

```csharp
public double[] GetBreakpoints()
Returns a copy of the breakpoints.
```
integral

virtual public double Integral(double a, double b)
Returns the value of an integral of the spline.

**Description**

Class **CsShape** computes a cubic spline interpolant to \( n \) data points \( x_i, f_i \) for \( i = 0, \ldots, n - 1 \). For ease of explanation, we will assume that \( x_i < x_{i+1} \), although it is not necessary for the user to sort these data values. If the data are strictly convex, then the computed spline is convex, \( C^2 \), and minimizes the expression

\[
\int_{x_1}^{x_n} (g'')^2
\]

over all convex \( C^1 \) functions that interpolate the data. In the general case when the data have both convex and concave regions, the convexity of the spline is consistent with the data and the above integral is minimized under the appropriate constraints. For more information on this interpolation scheme, we refer the reader to Micchelli et al. (1985) and Irvine et al. (1986).

One important feature of the splines produced by this class is that it is not possible, a priori, to predict the number of breakpoints of the resulting interpolant. In most cases, there will be breakpoints at places other than data locations. The method is nonlinear; and although the interpolant is a piecewise cubic, cubic polynomials are not reproduced. However, linear polynomials are reproduced.) This routine should be used when it is important to preserve the convex and concave regions implied by the data.

**Example: The shape preserving cubic spline interpolant**

A cubic spline interpolant to a function is computed consistent with the concavity of the data. The spline value at 0.05 is printed.

```csharp
using System;
using Imsl.Math;

class CsShapeEx1
{
    static void Main(String[] args)
    {
        double[] x = new double[0.00, 0.10, 0.20, 0.30, 0.40,
                             0.50, 0.60, 0.80, 1.00];
        double[] y = new double[0.00, 0.90, 0.95, 0.90, 0.10,
                             0.05, 0.05, 0.20, 1.00];

        CsShape cs = new CsShape(x, y);

        // ...}
```
double csv = cs.Eval(0.05);
Console.Out.WriteLine("The computed cubic spline value at " +
"point .05 is " + csv);
}
}

Output

The computed cubic spline value at point .05 is 0.55823122286482

CsSmooth Class

Extension of the Spline class to construct a smooth cubic spline from noisy data points.

public class Imsl.Math.CsSmooth : Spline

Constructors

CsSmooth
public CsSmooth(double[] xData, double[] yData)
Constructs a smooth cubic spline from noisy data using cross-validation to estimate the
smoothing parameter. All of the points have equal weights.

xData  A double array containing the x-coordinates of the data. Values must be
distinct.

yData  A double array containing the y-coordinates of the data. The arrays xData and
yData must have the same length.

CsSmooth
public CsSmooth(double[] xData, double[] yData, double[] weight)
Constructs a smooth cubic spline from noisy data using cross-validation to estimate the
smoothing parameter. Weights are supplied by the user.

xData  A double array containing the x-coordinates of the data. Values must be
distinct.

yData  A double array containing the y-coordinates of the data. The arrays xData and
yData must have the same length.

weight  A double array containing the relative weights. This array must have the same
length as xData.
Methods

Derivative
  virtual public double[] Derivative(double[] x, int ideriv)
  Returns the value of the derivative of the spline at each point of an array.

Derivative
  virtual public double Derivative(double x, int ideriv)
  Returns the value of the derivative of the spline at a point.

Derivative
  virtual public double Derivative(double x)
  Returns the value of the first derivative of the spline at a point.

Eval
  virtual public double[] Eval(double[] x)
  Returns the value of the spline at each point of an array.

Eval
  virtual public double Eval(double x)
  Returns the value of the spline at a point.

GetBreakpoints
  public double[] GetBreakpoints()
  Returns a copy of the breakpoints.

Integral
  virtual public double Integral(double a, double b)
  Returns the value of an integral of the spline.
Description

Class **CsSmooth** is designed to produce a $C^2$ cubic spline approximation to a data set in which the function values are noisy. This spline is called a smoothing spline. It is a natural cubic spline with knots at all the data abscissas $x = xData$, but it does not interpolate the data $(x_i, f_i)$. The smoothing spline $S$ is the unique $C^2$ function that minimizes

$$
\int_a^b S''(x)^2 \, dx
$$

subject to the constraint

$$
\sum_{i=0}^{n-1} |(S(x_i) - f_i)w_i|^2 \leq \sigma
$$

where $\sigma$ is the smoothing parameter. The reader should consult Reinsch (1967) for more information concerning smoothing splines. **CsSmooth** solves the above problem when the user provides the smoothing parameter $\sigma$. **CsSmoothC2** attempts to find the "optimal" smoothing parameter using the statistical technique known as cross-validation. This means that (in a very rough sense) one chooses the value of $\sigma$ so that the smoothing spline ($S_\sigma$) best approximates the value of the data at $x_i$, if it is computed using all the data except the $i$-th; this is true for all $i = 0, \ldots, n - 1$. For more information on this topic, we refer the reader to Craven and Wahba (1979).

**Example: The cubic spline interpolant to noisy data**

A cubic spline interpolant to noisy data is computed using cross-validation to estimate the smoothing parameter. The value of the spline at point 0.3010 is printed.

```csharp
using System;
using Imsl.Math;
using Imsl.Stat;

public class CsSmoothEx1
{
    public static void Main(String[] args)
    {
        int n = 300;
        double[] x = new double[n];
        double[] y = new double[n];
        for (int k = 0; k < n; k++)
        {
            x[k] = (3.0 * k) / (n - 1);
            y[k] = 1.0 / (0.1 + System.Math.Pow(3.0 * (x[k] - 1.0), 4));
        }
    }
}
```
// Seed the random number generator
rn.Multipier = 16807;

// Contaminate the data
for (int i = 0; i < n; i++)
{
    y[i] += 2.0 * (float) rn.NextDouble() - 1.0;
}

// Smooth the data
CsSmooth cs = new CsSmooth(x, y);
double csv = cs.Eval(0.3010);
Console.Out.WriteLine("The computed cubic spline value at "+
    "point .3010 is "+ csv);
}

Output
The computed cubic spline value at point .3010 is 0.0101201298963992

CsSmoothC2 Class
Extension of the Spline class used to construct a spline for noisy data points using an alternate method.

public class Imsl.Math.CsSmoothC2 : Spline

Constructors

CsSmoothC2
public CsSmoothC2(double[] xData, double[] yData, double sigma)
Constructs a smooth cubic spline from noisy data using an algorithm based on Reinsch (1967). All of the points have equal weights.

xData A double array containing the x-coordinates of the data. Values must be distinct.
yData A double array containing the y-coordinates of the data. The arrays xData and yData must have the same length.
sigma A double value specifying the smoothing parameter. sigma must not be negative.
CsSmoothC2

```csharp
public CsSmoothC2(double[] xData, double[] yData, double[] weight, double sigma)

Constructs a smooth cubic spline from noisy data using an algorithm based on Reinsch (1967) with weights supplied by the user.

**xData**  A double array containing the x-coordinates of the data. Values must be distinct.

**yData**  A double array containing the y-coordinates of the data. The arrays `xData` and `yData` must have the same length.

**weight**  A double array containing the weights. The arrays `xData` and `weight` must have the same length.

**sigma**  A double value specifying the smoothing parameter. `sigma` must not be negative.
```

## Methods

**Derivative**

```csharp
virtual public double[] Derivative(double[] x, int ideriv)

Returns the value of the derivative of the spline at each point of an array.
```

**Derivative**

```csharp
virtual public double Derivative(double x, int ideriv)

Returns the value of the derivative of the spline at a point.
```

**Derivative**

```csharp
virtual public double Derivative(double x)

Returns the value of the first derivative of the spline at a point.
```

**Eval**

```csharp
virtual public double[] Eval(double[] x)

Returns the value of the spline at each point of an array.
```

**Eval**

```csharp
virtual public double Eval(double x)

Returns the value of the spline at a point.
```
GetBreakpoints
    public double[] GetBreakpoints()
    Returns a copy of the breakpoints.

Integral
    virtual public double Integral(double a, double b)
    Returns the value of an integral of the spline.

Description

Class CsSmoothC2 is designed to produce a $C^2$ cubic spline approximation to a data set in which the function values are noisy. This spline is called a smoothing spline. It is a natural cubic spline with knots at all the data abscissas $x$, but it does not interpolate the data $(x_i, f_i)$. The smoothing spline $S_\sigma$ is the unique $C^2$ function that minimizes

$$
\int_a^b s'_\sigma(x)^2 \, dx
$$

subject to the constraint

$$
\sum_{i=0}^{n-1} |s_\sigma(x_i) - f_i|^2 \leq \sigma
$$

Recommended values for $\sigma$ depend on the weights, $w$. If an estimate for the standard deviation of the error in the $y$-values is available, then $w_i$ should be set to this value and the smoothing parameter should be chosen in the confidence interval corresponding to the left side of the above inequality. That is,

$$
n - \sqrt{2n} \leq \sigma \leq n + \sqrt{2n}
$$

CsSmoothC2 is based on an algorithm of Reinsch (1967). This algorithm is also discussed in de Boor (1978, pages 235-243).

Example: The cubic spline interpolant to noisy data with supplied weights

A cubic spline interpolant to noisy data is computed using supplied weights and smoothing parameter. The value of the spline at point 0.3010 is printed.
using System;
using Imsl.Math;
using Imsl.Stat;

public class CsSmoothC2Ex1
{
    public static void Main(String[] args)
    {
        // Set up a grid
        int n = 300;
        double[] x = new double[n];
        double[] y = new double[n];
        for (int k = 0; k < n; k++)
        {
            x[k] = 3.0 * ((double) (k) / (double) (n - 1));
            y[k] = 1.0 / (1.0 + System.Math.Pow(3.0 * (x[k] - 1.0), 4));
        }

        // Seed the random number generator
        rn.Multiplier = 16807;

        // Contaminate the data
        for (int i = 0; i < n; i++)
        {
            y[i] = y[i] + 2.0 * (float) rn.NextDouble() - 1.0;
        }

        // Set the weights
        double sdev = 1.0 / System.Math.Sqrt(3.0);
        double[] weights = new double[n];
        for (int i = 0; i < n; i++)
        {
            weights[i] = sdev;
        }

        // Set the smoothing parameter
        double smpar = (double) n;

        // Smooth the data
        CsSmoothC2 cs = new CsSmoothC2(x, y, weights, smpar);
        double csv = cs.Eval(0.3010);
        Console.Out.WriteLine("The computed cubic spline value at " +
            "point .3010 is " + csv);
    }
}

Output
The computed cubic spline value at point .3010 is 0.0335028881575695
BSpline Class

Spline represents and evaluates univariate B-splines.

public class Imsl.Math.BSpline

Constructor

BSpline

BSpline()

Initializes a new instance of the Imsl.Math.BSpline (p. 57) class.

Methods

Derivative

public double Derivative(double x)

Returns the value of the first derivative of the B-spline at a point.

x  A double which specifies the point at which the derivative is to be evaluated.

Returns — A double containing the value of the first derivative of the B-spline at the point x.

Derivative

public double Derivative(double x, int ideriv)

Returns the value of the derivative of the B-spline at a point.

If ideriv is zero, the function value is returned. If one, the first derivative is returned, etc.

x  A double which specifies the point at which the derivative is to be evaluated.

ideriv  A int specifying the derivative to be computed.

Returns — A double containing the value of the derivative of the B-spline at the point x.

Derivative

public double[] Derivative(double[] x, int ideriv)

Returns the value of the derivative of the B-spline at each point of an array.

If ideriv is zero, the function value is returned. If one, the first derivative is returned, etc.

x  A double array of points at which the derivative is to be evaluated.

ideriv  A int specifying the derivative to be computed.

Returns — A double array containing the value of the derivative the B-spline at each point of the array x.
Eval
  public double Eval(double x)
  Returns the value of the B-spline at a point.

  x A double which specifies the point at which the B-spline is to be evaluated.

  Returns — A double giving the value of the B-spline at the point x.

Eval
  public double[] Eval(double[] x)
  Returns the value of the B-spline at each point of an array.

  x A double array of points at which the B-spline is to be evaluated.

  Returns — A double array containing the value of the B-spline at each point of the array x.

GetKnots
  public double[] GetKnots()
  Returns a copy of the knot sequence.

  Returns — A double array containing a copy of the knot sequence.

GetSpline
  public Imsl.Math.Spline GetSpline()
  Returns a Spline representation of the B-spline.

  Returns — A Spline representation of the B-spline.

Integral
  public double Integral(double a, double b)
  Returns the value of an integral of the B-spline.

  a A double specifying the lower limit of integration.

  b A double specifying the upper limit of integration.

  Returns — A double which specifies the integral of the B-spline from a to b.

Description

B-splines provide a particularly convenient and suitable basis for a given class of smooth ppoly functions. Such a class is specified by giving its breakpoint sequence, its order $k$, and the required smoothness across each of the interior breakpoints. The corresponding B-spline basis is specified by giving its knot sequence $t \in \mathbb{R}^M$. The specification rule is as follows: If the class is to have all derivatives up to and including the $j$-th derivative continuous across the interior breakpoint $\xi_i$, then the number $\xi_i$ should occur $k - j - 1$ times in the knot sequence. Assuming that $\xi_1$ and $\xi_n$ are the endpoints of the interval of interest, choose the first $k$ knots equal to $\xi_1$ and the last $k$ knots equal to $\xi_n$. This can be done because the B-splines are defined to be right continuous near $\xi_1$ and left continuous near $\xi_n$. 

58 • BSpine Class  IIMLS C# Numerical Library
When the above construction is completed, a knot sequence \( t \) of length \( M \) is generated, and there are \( m = M-k \) B-splines of order \( k \), for example \( B_0, \ldots, B_{m-1} \), spanning the \( ppoly \) functions on the interval with the indicated smoothness. That is, each \( ppoly \) function in this class has a unique representation \( p = a_0 B_0 + a_1 B_1 + \ldots + a_{m-1} B_{m-1} \) as a linear combination of B-splines. A B-spline is a particularly compact \( ppoly \) function. \( B_i \) is a nonnegative function that is nonzero only on the interval \( [t_i, t_{i+k}] \). More precisely, the support of the \( i \)-th B-spline is \( [t_i, t_{i+k}] \). No \( ppoly \) function in the same class (other than the zero function) has smaller support (i.e., vanishes on more intervals) than a B-spline. This makes B-splines particularly attractive basis functions since the influence of any particular B-spline coefficient extends only over a few intervals.

**Example: The B-spline interpolant**

A B-Spline interpolant to data is computed. The value of the spline at point .23 is printed.

```csharp
using System;
using Imsl.Math;

public class BsInterpolateEx1
{
    public static void Main(String[] args)
    {
        int n = 11;
        double[] x = new double[n];
        double[] y = new double[n];

        double h = 2.0 * System.Math.PI / 15.0 / 10.0;
        for (int k = 0; k < n; k++)
        {
            x[k] = h * (double) (k);
            y[k] = System.Math.Sin(15.0 * x[k]);
        }

        BsInterpolate bs = new BsInterpolate(x, y);
        double bsv = bs.Eval(0.23);
        Console.Out.WriteLine("The computed B-spline value at point ", 0.23 is "+ bsv);
    }
}
```

**Output**

The computed B-spline value at point .23 is -0.303418399276769

**Example: The B-spline least squares fit**

A B-Spline least squares fit to data is computed. The value of the spline at point 4.5 is printed.
using System;
using Imsl.Math;

public class BsLeastSquaresEx1
{
    public static void Main(string[] args)
    {
        double[] x = new double[]{0, 1, 2, 3, 4, 5, 8, 9, 10};
        double[] y = new double[]{1.0, 0.8, 2.4, 3.1, 4.5, 5.8, 6.2, 4.9, 3.7};

        BsLeastSquares bs = new BsLeastSquares(x, y, 5);
        double bsv = bs.Eval(4.5);
        Console.Out.WriteLine("The computed B-spline value at point " +
                              "4.5 is " + bsv);
    }
}

Output

The computed B-spline value at point 4.5 is 5.22855432359694

---

**BsInterpolate Class**

Extension of the BSpline class to interpolate data points.

public class Imsl.Math.BsInterpolate : BSpline

**Constructors**

BsInterpolate

public BsInterpolate(double[] xData, double[] yData)
Constructs a B-spline that interpolates the given data points. The computed B-spline will be order 4 (cubic) and have a default "not-a-knot" spline knot sequence.

xData  A double array containing the x-coordinates of the data. Values must be distinct.

yData  A double array containing the y-coordinates of the data. The arrays xData and yData must have the same length.

BsInterpolate

public BsInterpolate(double[] xData, double[] yData, int order)
Constructs a B-spline that interpolates the given data points and order, using a default "not-a-knot" spline knot sequence.
**xData**  A double array containing the x-coordinates of the data. Values must be distinct.

**yData**  A double array containing the y-coordinates of the data. The arrays xData and yData must have the same length.

**order**  An int denoting the order of the B-spline.

**BsInterpolate**

```
public BsInterpolate(double[] xData, double[] yData, int order, double[] knot)
```

Constructs a B-spline that interpolates the given data points, using the specified order and knots.

**xData**  A double array containing the x-coordinates of the data. Values must be distinct.

**yData**  A double array containing the y-coordinates of the data. The arrays xData and yData must have the same length.

**order**  A int denoting the order of the spline.

**knot**  A double array containing the knot sequence for the B-spline.

### Methods

**Derivative**

```
public double Derivative(double x)
```

Returns the value of the first derivative of the B-spline at a point.

**Derivative**

```
public double Derivative(double x, int ideriv)
```

Returns the value of the derivative of the B-spline at a point.

**Derivative**

```
public double[] Derivative(double[] x, int ideriv)
```

Returns the value of the derivative of the B-spline at each point of an array.

**Eval**

```
public double Eval(double x)
```

Returns the value of the B-spline at a point.
Eval

public double[] Eval(double[] x)
Returns the value of the B-spline at each point of an array.

GetKnots

public double[] GetKnots()
Returns a copy of the knot sequence.

GetSpline

public Imsl.Math.Spline GetSpline()
Returns a Spline representation of the B-spline.

Integral

public double Integral(double a, double b)
Returns the value of an integral of the B-spline.

Description

Given the data points $x = x_{\text{Data}}$, $f = y_{\text{Data}}$, and $n$ the number of elements in $x_{\text{Data}}$ and $y_{\text{Data}}$, the default action of BsInterpolate computes a cubic (order = 4) spline interpolant $s$ to the data using a default "not-a-knot" knot sequence. Constructors are also provided that allow the order and knot sequence to be specified. This algorithm is based on the routine SPLINT by de Boor (1978, p. 204).

First, the xData vector is sorted and the result is stored in $x_i$. The elements of yData are permuted appropriately and stored in $f_i$, yielding the equivalent data $(x_i, f_i)$ for $i = 0$ to $n-1$.

The following preliminary checks are performed on the data, with $k = \text{order}$. We verify that

- $x_i < x_{i+1}$ for $i = 0, \ldots, n-2$
- $t_i < t_{i+k}$ for $i = 0, \ldots, n-1$
- $t_i < t_{i+1}$ for $i = 0, \ldots, n+k-2$

The first test checks to see that the abscissas are distinct. The second and third inequalities verify that a valid knot sequence has been specified.

In order for the interpolation matrix to be nonsingular, we also check $t_{k-1} \leq x_i \leq t_n$ for $i = 0$ to $n-1$. This first inequality in the last check is necessary since the method used to generate the entries of the interpolation matrix requires that the $k$ possibly nonzero B-splines at $x_i$, $B_{j-k+1}, \ldots, B_j$, where $j$ satisfies $t_j \leq x_i < t_{j+1}$ be well-defined (that is, $j - k + 1 \geq 0$).

Example: The B-spline interpolant

A B-Spline interpolant to data is computed. The value of the spline at point .23 is printed.
using System;
using Imsl.Math;

class BsInterpolateEx1
{
    public static void Main(String[] args)
    {
        int n = 11;
        double[] x = new double[n];
        double[] y = new double[n];

        double h = 2.0 * System.Math.PI / 15.0 / 10.0;
        for (int k = 0; k < n; k++)
        {
            x[k] = h * (double) (k);
            y[k] = System.Math.Sin(15.0 * x[k]);
        }

        BsInterpolate bs = new BsInterpolate(x, y);
        double bsv = bs.Eval(0.23);
        Console.Out.WriteLine("The computed B-spline value at point ".23 is "+ bsv);
    }
}

Output
The computed B-spline value at point .23 is -0.303418399276769

BsLeastSquares Class

Extension of the BSpline class to compute a least squares spline approximation to data points.

class Imsl.Math.BsLeastSquares : BSpline

Constructors

BsLeastSquares

public BsLeastSquares(double[] xData, double[] yData, int nCoef)
Constructs a least squares B-spline approximation to the given data points.

xData A double array containing the x-coordinates of the data.
yData A double array containing the y-coordinates of the data. The arrays xData and yData must have the same length.
nCoef  A int denoting the linear dimension of the spline subspace. It should be smaller than the number of data points and greater than or equal to the order of the spline (whose default value is 4).

BsLeastSquares
public BsLeastSquares(double[] xData, double[] yData, int nCoef, int order)
Constructs a least squares B-spline approximation to the given data points.

xData  A double array containing the x-coordinates of the data.
yData  A double array containing the y-coordinates of the data. The arrays xData and yData must have the same length.
nCoef  A int denoting the linear dimension of the spline subspace. It should be smaller than the number of data points and greater than or equal to the order of the spline.
order  A int denoting the order of the spline.

BsLeastSquares
public BsLeastSquares(double[] xData, double[] yData, int nCoef, int order, double[] weight, double[] knot)
Constructs a least squares B-spline approximation to the given data points.

xData  A double array containing the x-coordinates of the data.
yData  A double array containing the y-coordinates of the data. The arrays xData and yData must have the same length.
nCoef  A int denoting the linear dimension of the spline subspace. It should be smaller than the number of data points and greater than or equal to the order of the spline.
order  A int denoting the order of the spline.
weight  A double array containing the weights for the data. The arrays xData, yData and weight must have the same length.
knot  A double array containing the knot sequence for the spline.

Methods

Derivative
public double Derivative(double x)
Returns the value of the first derivative of the B-spline at a point.

Derivative
public double Derivative(double x, int ideriv)
Returns the value of the derivative of the B-spline at a point.
Derivative

public double[] Derivative(double[] x, int ideriv)
Returns the value of the derivative of the B-spline at each point of an array.

Eval

public double Eval(double x)
Returns the value of the B-spline at a point.

Eval

public double[] Eval(double[] x)
Returns the value of the B-spline at each point of an array.

GetKnots

public double[] GetKnots()
Returns a copy of the knot sequence.

GetSpline

public Imsl.Math.Spline GetSpline()
Returns a Spline representation of the B-spline.

Integral

public double Integral(double a, double b)
Returns the value of an integral of the B-spline.

Description

Let’s make the identifications

\[n = \text{xData.length}\]
\[x = \text{xData}\]
\[f = \text{yData}\]
\[m = \text{nCoef}\]
\[k = \text{order}\]

For convenience, we assume that the sequence \(x\) is increasing, although the class does not require this.
By default, \( k = 4 \), and the knot sequence we select equally distributes the knots through the distinct \( x_i \)'s. In particular, the \( m + k \) knots will be generated in \([x_1, x_n]\) with \( k \) knots stacked at each of the extreme values. The interior knots will be equally spaced in the interval.

Once knots \( t \) and weights \( w \) are determined, then the spline least-squares fit to the data is computed by minimizing over the linear coefficients \( a_j \)

\[
\sum_{i=0}^{n-1} w_i \left[ f_i - \sum_{j=1}^{m} a_j B_j(x_i) \right]^2
\]

where the \( B_j, j = 1, ..., m \) are a (B-spline) basis for the spline subspace.

This algorithm is based on the routine L2APPR by deBoor (1978, p. 255).

**Example: The B-spline least squares fit**

A B-Spline least squares fit to data is computed. The value of the spline at point 4.5 is printed.

```csharp
using System;
using Imsl.Math;

public class BsLeastSquaresEx1
{
    public static void Main(String[] args)
    {
        double[] x = new double[]{0, 1, 2, 3, 4, 5, 8, 9, 10};
        double[] y = new double[]{1.0, 0.8, 2.4, 3.1, 4.5, 5.8, 6.2, 4.9, 3.7};

        BsLeastSquares bs = new BsLeastSquares(x, y, 5);
        double bsv = bs.Eval(4.5);
        Console.Out.WriteLine("The computed B-spline value at point " +
                           "4.5 is " + bsv);
    }
}
```

**Output**

The computed B-spline value at point 4.5 is 5.22855432359694

---

**RadialBasis Class**

Computes a least-squares fit to scattered data.
public class Imsl.Math.RadialBasis

Properties

ANOVA
public Imsl.Stat.ANOVA ANOVA {get; }
Returns the ANOVA statistics from the linear regression. An ANOVA table and related statistics.
See Also: Imsl.Stat.LinearRegression (p. 297), Imsl.Stat.ANOVA (p. 341)

RadialFunction
public Imsl.Math.RadialBasis.IFunction RadialFunction {get; set; }
The radial function. Radial basis function.

Constructor

RadialBasis
public RadialBasis(int nDim, int nCenters)
Creates a new instance of RadialBasis.

nDim An int specifying the number of dimensions.
nCenters An int specifying the number of centers.

Methods

Eval
public double Eval(double[] x)
Returns the value of the radial basis approximation at a point.

x A double array containing the location of the data point at which the approximation is to be computed.

Returns — The value of the radial basis approximation at x.

Eval
public double[] Eval(double[,] x)
Returns the value of the radial basis approximation at a point.

x A double[,], the point at which the radial basis is to be evaluated.

Returns — A double[] giving the value of the radial basis at the point x.

Gradient
public double[] Gradient(double[] x)
Returns the gradient of the radial basis approximation at a point.
A double array containing the location of the data point at which the approximation’s gradient is to be computed.

Returns — A double array, of length nDim containing the value of the gradient of the radial basis approximation at x.

Update

public void Update(double[] x, double f)
Adds a data point with weight = 1.

x A double array containing the location of the data point.
f A double containing the function value at the data point.

Update

public void Update(double[] x, double f, double w)
Adds a data point with a specified weight.

x A double array containing the location of the data point.
f A double containing the function value at the data point.
w A double containing the weight of this data point.

Update

public void Update(double[,] x, double[] f)
Adds a set of data points, all with weight = 1.

x A double matrix of size nPoints by nDim containing the location of the data points.
f A double array containing the function values at the data points.

Update

public void Update(double[,] x, double[] f, double[] w)
Adds a set of data points with user-specified weights.

x A double matrix of size nPoints by nDim containing the location of the data points.
f A double array containing the function values at the data points.
w A double array containing the weights associated with the data points.

Description

RadialBasis computes a least-squares fit to scattered data in \( \mathbb{R}^d \), where \( d \) is the dimension. More precisely, we are given data points

\[
x_0, \ldots, x_{n-1} \in \mathbb{R}^d
\]

and function values

\[
f_0, \ldots, f_{n-1} \in \mathbb{R}^1
\]
The radial basis fit to the data is a function $F$ which approximates the above data in the sense that it minimizes the sum-of-squares error

$$
\sum_{i=0}^{n-1} w_i (F(x_i) - f_i)^2
$$

where $w$ are the weights. Of course, we must restrict the functional form of $F$. Here we assume it is a linear combination of radial functions:

$$
F(x) = \sum_{j=0}^{m-1} \alpha_j \phi(||x - c_j||)
$$

The $c_j$ are the centers.

A radial function, $\phi(r)$, maps $[0, \infty)$ into $\mathbb{R}^1$. The default radial function is the Hardy multiquadric,

$$
\phi(r) \equiv \sqrt{r^2 + \delta^2}
$$

with $\delta = 1$. An alternate radial function is the Gaussian, $e^{-ax^2}$.

By default, the centers are points in a Faure sequence, scaled to cover the box containing the data.

**Example: Radial Basis Function Approximation**

The function

$$
e^{-\|\vec{x}\|^2/d}
$$

where $d$ is the dimension, is evaluated at a set of randomly chosen points. Random noise is added to the values and a radial basis approximated to the noisy data is computed. The radial basis fit is then compared to the original function at another set of randomly chosen points. Both the average error and the maximum error are computed and printed.

In this example, the dimension $d=10$. The function is sampled at 200 random points, in the $[-1,1]^d$ cube, to which what noise in the range $[-0.2,0.2]$ is added. The error is computed at 1000 random points, also from the $[-1,1]^d$ cube. The compute errors are less than the added noise.

```csharp
using System;
using Imsl.Math;

class RadialBasisEx1
{
    public static void Main(String[] args)
    {
        int nDim = 10;

        // Sample, with noise, the function at 100 randomly choosen points
        int nData = 200;
        double[,] xData = new double[nData,nDim];
```
```csharp
double[] fData = new double[nData];
double[] tmp = new double[nDim];
for (int k = 0; k < nData; k++)
{
    for (int i = 0; i < nDim; i++)
    {
        tmp[i] = xData[k,i] = 2.0 * rand.NextDouble() - 1.0;
    }
    // noisy sample
    fData[k] =
        fcn(tmp) + 0.20 * (2.0 * rand.NextDouble() - 1.0);
}

// Compute the radial basis approximation using 25 centers
int nCenters = 25;
RadialBasis rb = new RadialBasis(nDim, nCenters);
rb.Update(xData, fData);

// Compute the error at a randomly selected set of points
int nTest = 1000;
double maxError = 0.0;
double aveError = 0.0;
double[] x = new double[nDim];
for (int k = 0; k < nTest; k++)
{
    for (int i = 0; i < nDim; i++)
    {
        x[i] = 2.0 * rand.NextDouble() - 1.0;
    }
    double error = System.Math.Abs(fcn(x) - rb.Eval(x));
    aveError += error;
    maxError = System.Math.Max(error, maxError);
    double f = fcn(x);
}
aveError /= nTest;
Console.Out.WriteLine("average error is " + aveError);
Console.Out.WriteLine("maximum error is " + maxError);

// The function to approximate
internal static double fcn(double[] x)
{
    double sum = 0.0;
    for (int k = 0; k < x.Length; k++)
    {
        sum += x[k] * x[k];
    }
    sum /= x.Length;
    return System.Math.Exp(-sum);
}
```

---

70 • RadialBasis Class  
IMSL C# Numerical Library
average error is 0.041978979550254
maximum error is 0.17166811944546

RadialBasis.IFunction Interface

Public interface for the user supplied function to the RadialBasis object.

public interface Imsl.Math.RadialBasis.IFunction

Methods

F
abstract public double F(double x)
A radial basis function.

x  A double, the point at which the function is to be evaluated.

Returns — A double, the value of the function at x.

G
abstract public double G(double x)
The derivative of the radial basis function.

x  A double, the point at which the function is to be evaluated.

Returns — A double, the value of the function at x.

RadialBasis.Gaussian Class

The Gaussian basis function, $e^{-ax^2}$.

public class Imsl.Math.RadialBasis.Gaussian implements

Constructor

Gaussian
    public Gaussian(double a)
The Gaussian basis function, $e^{-ax^2}$.
a  A double, the value of the function at x

Methods

F
Final public double F(double x)
A radial basis function.

x  A double, the point at which the function is to be evaluated.

Returns — A double, the value of the function at x.

G
Final public double G(double x)
The derivative of the radial basis function.

x  A double, the point at which the function is to be evaluated.

Returns — A double, the value of the function at x.

RadialBasis.HardyMultiquadric Class

The Hardy multiquadric basis function, $\sqrt{r^2 + \delta^2}$.

public class Imsl.Math.RadialBasis.HardyMultiquadric implements

Constructor

HardyMultiquadric

public HardyMultiquadric(double delta)
Creates a Hardy multiquadric basis function.

    delta  The parameter in the function definition.

Methods

F
Final public double F(double x)
A radial basis function.

x  A double, the point at which the function is to be evaluated.

Returns — A double, the value of the function at x.
Final public double G(double x)
The derivative of the radial basis function.

x  A double, the point at which the function is to be evaluated.

Returns — A double, the value of the function at x.
Chapter 4: Quadrature

Types

class Quadrature .......................................................... 76
interface Quadrature.IFunction ........................................... 82
class HyperRectangleQuadrature ........................................... 82
interface HyperRectangleQuadrature.IFunction .......................... 85

Usage Notes

Univariate Quadrature

Class Quadrature computes approximations to integrals of the form

\[ \int_{c}^{b} f(x) \, dx \]

Quadrature computes an estimated answer \( R \). An optional value \( \text{ErrorEstimate} = E \) estimates the error. These numbers are related as follows:

\[ \left| \int_{a}^{b} f(x) \, dx - R \right| \leq E \leq \max \left\{ \epsilon, \rho \left| \int_{a}^{b} f(x) \, dx \right| \right\} \]

One situation that occasionally arises in univariate quadrature concerns the approximation of integrals when only tabular data are given. The functions described above do not directly address this question. However, the standard method for handling this problem is first to interpolate the data, and then to integrate the interpolant. This can be accomplished by using a IMSL C# Library spline interpolation class derived from Imsl.Math.Spline and the method Imsl.Math.Spline.Integral (a,b)
**Multivariate Quadrature**

The class `HypercubeQuadrature` computes an approximation to the integral of a function of \( n \) variables over a hyper-rectangle.

\[
\int_{a_1}^{b_1} \ldots \int_{a_n}^{b_n} f(x_1, \ldots, x_n)dx_n \ldots dx_1
\]

---

**Quadrature Class**

*Quadrature* is a general-purpose integrator that uses a globally adaptive scheme in order to reduce the absolute error.

```csharp
public class Imsl.Math.Quadrature
```

**Properties**

AbsoluteError

```csharp
   public double AbsoluteError {get; set; }
   The absolute error tolerance. A double scalar value specifying the absolute error.
```

ErrorEstimate

```csharp
   public double ErrorEstimate {get; }
   Returns an estimate of the relative error in the computed result. A double specifying an estimate of the relative error in the computed result.
```

ErrorStatus

```csharp
   public int ErrorStatus {get; }
   Returns the non-fatal error status. An int specifying the non-fatal error status:
```
<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Maximum number of subdivisions allowed has been achieved. One can allow more subdivisions by setting the <code>MaxSubintervals</code>. If this yields no improvement it is advised to analyze the integrand in order to determine the integration difficulties. If the position of a local difficulty can be determined (e.g. singularity, discontinuity within the interval) one will probably gain from splitting up the interval at this point and calling the integrator on the subranges. If possible, an appropriate special-purpose integrator should be used, which is designed for handling the type of difficulty involved.</td>
</tr>
<tr>
<td>2</td>
<td>The occurrence of roundoff error is detected, which prevents the requested tolerance from being achieved. The error may be under-estimated.</td>
</tr>
<tr>
<td>3</td>
<td>Extremely bad integrand behavior occurs at some points of the integration interval.</td>
</tr>
<tr>
<td>5</td>
<td>The algorithm does not converge. Roundoff error is detected in the extrapolation table. It is presumed that the requested tolerance cannot be achieved, and that the returned result is the best which can be obtained.</td>
</tr>
<tr>
<td>6</td>
<td>The integral is probably divergent, or slowly convergent. It must be noted that divergence can occur with any other status value.</td>
</tr>
</tbody>
</table>

**Extrapolation**

```csharp
public bool Extrapolation {get; set; }
```

If `true`, the epsilon-algorithm for extrapolation is enabled. A `boolean`, `true` if the epsilon-algorithm for extrapolation is to be enabled, `false` otherwise.

The default is `false` (extrapolation is not used).

**MaxSubintervals**

```csharp
public int MaxSubintervals {get; set; }
```

The maximum number of subintervals allowed. An `int` specifying the maximum number of subintervals to be allowed.

The default value is 500.

**RelativeError**

```csharp
public double RelativeError {get; set; }
```

The relative error tolerance. A `double` scalar value specifying the relative error.

**Rule**

```csharp
public int Rule {get; set; }
```

The Gauss-Kronrod rule. An `int` specifying the rule to be used.

The default is 3.
### Constructor

Quadrature

```java
public Quadrature()
    Constructs a Quadrature object.
```

### Method

**Eval**

```java
public double Eval(Imsl.Math.Quadrature.IFunction f, double a, double b)
    Returns the value of the integral from a to b.
```

- **f** The function to be integrated.
- **a** A double specifying the lower limit of integration.
- **b** A double specifying the upper limit of integration, either or both of a and b can be `Double.POSITIVE_INFINITY` or `Double.NEGATIVE_INFINITY`.

Returns — A double specifying the integral value from a to b.

### Description

*Quadrature* subdivides the interval \([A, B]\) and uses a \((2k + 1)\)-point Gauss-Kronrod rule to estimate the integral over each subinterval. The error for each subinterval is estimated by comparison with the \(k\)-point Gauss quadrature rule. The subinterval with the largest estimated error is then bisected and the same procedure is applied to both halves. The bisection process is continued until either the error criterion is satisfied, roundoff error is detected, the subintervals become too small, or the maximum number of subintervals allowed is reached. This class is based on the subroutine *QAG* by Piessens et al. (1983).

### Example 1: Integral \( \int_1^3 e^{2x} \, dx \)

The integral \( \int_1^3 e^{2x} \, dx \) is computed and compared to its expected value.

```csharp
using System;
```
using Imsl.Math;

public class QuadratureEx1 : Quadrature.IFunction
{
    public double F(double x)
    {
        return Math.Exp(2.0 * x);
    }

    public static void Main(String[] args)
    {
        Quadrature q = new Quadrature();
        Quadrature.IFunction fcn = new QuadratureEx1();
        double result = q.Eval(fcn, 1.0, 3.0);

        double expect =
            (System.Math.Exp(6) - System.Math.Exp(2)) / 2.0;
        Console.Out.WriteLine("result = " + result);
        Console.Out.WriteLine("expect = " + expect);
    }
}

Output

result = 198.019868696902
expect = 198.019868696902

Example 2: Integral \( \int_{0}^{\infty} e^{-x} \, dx \)

The integral \( \int_{0}^{\infty} e^{-x} \, dx \) is computed and compared to its expected value.

using System;
using Imsl.Math;

public class QuadratureEx2 : Quadrature.IFunction
{
    public double F(double x)
    {
        return Math.Exp(- x);
    }

    public static void Main(String[] args)
    {
        Quadrature q = new Quadrature();
        Quadrature.IFunction fcn = new QuadratureEx2();
        double result = q.Eval(fcn, 0.0, Double.PositiveInfinity);

        double expect = 1.0;
        Console.Out.WriteLine("result = " + result);
        Console.Out.WriteLine("expect = " + expect);
    }
}
Example 3: Integral of the entire real line

The integral \( \int_{-\infty}^{\infty} \frac{x}{4e^x + 9e^{-x}} \, dx \) is computed and compared to its expected value. This integral is evaluated in Gradshteyn and Ryzhik (equation 3.417.1).

```csharp
using System;
using Imsl.Math;

public class QuadratureEx3 : Quadrature.IFunction
{
    public double F(double x)
    {
        return x / (4.0 * Math.Exp(x) + 9.0 * Math.Exp(-x));
    }

    public static void Main(String[] args)
    {
        Quadrature q = new Quadrature();
        Quadrature.IFunction fcn = new QuadratureEx3();
        double result = q.Eval(fcn, Double.NegativeInfinity, Double.PositiveInfinity);
        double expect = System.Math.PI * System.Math.Log(1.5) / 12.0;
        Console.Out.WriteLine("result = " + result);
        Console.Out.WriteLine("expect = " + expect);
    }
}
```

Output

result = 0.106150517076628
expect = 0.106150517076633
### Example 4: Integral of an oscillatory function

The integral of $\cos(ax)$ for $a = 10^4$ is computed and compared to its expected value. Because the function is highly oscillatory, the quadrature rule is set to 6. The relative error tolerance is also set.

```csharp
using System;
using Imsl.Math;

class QuadratureEx4 : Quadrature.IFunction
{
    private double a;

    public QuadratureEx4(double a)
    {
        this.a = a;
    }

    public double F(double x)
    {
        return Math.Cos(a * x);
    }

    public static void Main(String[] args)
    {
        double a = 1.0e4;
        Quadrature.IFunction fcn = new QuadratureEx4(a);

        Quadrature q = new Quadrature();
        q.Rule = 6;
        q.RelativeError = 1e-10;
        double result = q.Eval(fcn, 0.0, 1.0);

        double expect = Math.Sin(a) / a;
        Console.Out.WriteLine("result = "+result);
        Console.Out.WriteLine("expect = "+expect);
        Console.Out.WriteLine("relative error estimate = "+q.ErrorEstimate);
    }
}
```
Output

result = -3.05614388902526E-05
expect = -3.05614388888252E-05
relative error = -4.67047941622356E-11
relative error estimate = 1.04883755414239E-08

---

**Quadrature.IFunction Interface**

Interface defining function for the Quadrature class.

```csharp
public interface Imsl.Math.Quadrature.IFunction

Method

F
    abstract public double F(double x)
    Function to be integrated.
    x - A double specifying the point at which the function is to be evaluated.
    Returns — A double specifying the value of the function at x.
```

---

**HyperRectangleQuadrature Class**

HyperRectangleQuadrature integrates a function over a hypercube.

```csharp
public class Imsl.Math.HyperRectangleQuadrature

Properties

AbsoluteError
    public double AbsoluteError {get; set; }
    Sets the absolute error tolerance. A double scalar value specifying the absolute error tolerance.

ErrorEstimate
    public double ErrorEstimate {get; }
    Returns an estimate of the relative error in the computed result. A double specifying an estimate of the relative error in the computed result.
```
RelativeError

    public double RelativeError {get; set; }
Sets the relative error tolerance. A double scalar value specifying the relative error
tolerance.

Constructors

HyperRectangleQuadrature

    public HyperRectangleQuadrature(int dimension)
Constructs a HyperRectangleQuadrature object.

dimension  A int which specifies the dimension of the Faure sequence.

HyperRectangleQuadrature

    public HyperRectangleQuadrature(IMSL.Stat.IRandomSequence sequence)
Constructs a HyperRectangleQuadrature object.

sequence  A IRandomSequence object containing the random number sequence.

Methods

Eval

    public double Eval(IMSL.Math.HyperRectangleQuadrature.IFunction f)
Returns the value of the integral over the unit cube.

f  A IFunction containing the function to be integrated.

Returns — A double containing the value of the integral over the unit cube.

Eval

    public double Eval(IMSL.Math.HyperRectangleQuadrature.IFunction f, double[] a, double[] b)
Returns the value of the integral over a cube.

f  A IFunction containing the function to be integrated.
a  A double specifying the lower limit of integration. If null all of the lower limits
default to 0.
b  A double specifying the upper limit of integration. If null all of the upper limits
default to 1.

Returns — A double containing the value of the integral over the unit cube.
Description

This class is used to evaluate integrals of the form:

\[
\int_{a_0}^{b_0} \cdots \int_{a_{n-1}}^{b_{n-1}} f(x_0, \ldots, x_{n-1}) \, dx_0 \cdots dx_{n-1}
\]

Integration of functions over hypercubes by Monte Carlo, in which the integral is evaluated as the value of the function averaged over a sequence of randomly chosen points. Under mild assumptions on the function, this method will converge like 1/√n, where n is the number of points at which the function is evaluated.

It is possible to improve on the performance of Monte Carlo by carefully choosing the points at which the function is to be evaluated. Randomly distributed points tend to be non-uniformly distributed. The alternative to a sequence of random points is a low-discrepancy sequence. A low-discrepancy sequence is one that is highly uniform.

This function is based on the low-discrepancy Faure sequence as computed by Imsl.Stat.FaureSequence (p. 588).

Example: HyperRectangle Quadrature

This example evaluates the following multidimensional integral, with n=10.

\[
\int_{a_0}^{b_0} \cdots \int_{a_{n-1}}^{b_{n-1}} \left[ \sum_{i=0}^{n} (-1)^i \prod_{j=0}^{i} x_j \right] \, dx_0 \cdots dx_{n-1} = \frac{1}{3} \left[ 1 - \left( -\frac{1}{2} \right)^n \right]
\]

using System;
using Imsl.Math;

public class HyperRectangleQuadratureEx1 :
  HyperRectangleQuadrature.IFunction
{
  public double F(double[] x)
  {
    int sign = 1;
    double sum = 0.0;
    for (int i = 0; i < x.Length; i++)
    {
      double prod = 1.0;
      for (int j = 0; j <= i; j++)
      {
        prod *= x[j];
      }
      sum += sign * prod;
      sign = - sign;
    }
    return sum;
  }
}
public static void Main(String[] args)
{
    HyperRectangleQuadrature q = new HyperRectangleQuadrature(10);
    double result = q.Eval(new HyperRectangleQuadratureEx1());
    Console.Out.WriteLine("result = " + result);
}

Output

result = 0.333125383208954

HyperRectangleQuadrature.IFunction Interface

Interface for the HyperRectangleQuadrature function.

public interface Imsl.Math.HyperRectangleQuadrature.IFunction

Method

F

abstract public double F(double[] x)
Returns the value of the function at the given point.

x   A double array specifying the point at which the function is to be evaluated.

Returns — A double specifying the value of the function at x.
Chapter 5: Differential Equations

Types

class OdeRungeKutta ................................................................. 88
interface OdeRungeKutta.IFunction ............................................. 91

Usage Notes

Ordinary Differential Equations

An ordinary differential equation is an equation involving one or more dependent variables called \( y_i \), one independent variable, \( t \), and derivatives of the \( y_i \) with respect to \( t \).

In the initial-value problem (IVP), the initial or starting values of the dependent variables \( y_i \) at a known value \( t = t_0 \) are given. Values of \( y_i(t) \) for \( t > 0 \) or \( t < t_0 \) are required.

The OdeRungeKutta class solves the IVP for ODEs of the form

\[
\frac{dy_i}{dt} = y'_i = f_i (t, y_1, \ldots, y_N) \quad i = 1, \ldots, N
\]

with \( y_i = (t = t_0) \) specified. Here, \( f_i \) is a user-supplied function that must be evaluated at any set of values \((t, y_1, \ldots, y_N), i = 1, \ldots, N\).

This problem statement is abbreviated by writing it as a system of first-order ODEs,

\[
y(t) [y_1(t), \ldots, y_N(t)]^T, [f_1(t, y), \ldots, f_N(t, y)]^T
\]

so that the problem becomes \( y' = f(t, y) \) with initial values \( y(t_0) \).

The system

\[
\frac{dy}{dt} = y' = f(t, y)
\]
is said to be *stiff* if some of the eigenvalues of the Jacobian matrix

\[ \left\{ \frac{\partial y'_i}{\partial y_j} \right\} \]

are large and negative. This is frequently the case for differential equations modeling the behavior of physical systems, such as chemical reactions proceeding to equilibrium where subspecies effectively complete their reactions in different epochs. An alternate model concerns discharging capacitors such that different parts of the system have widely varying decay rates (or *time constants*).

Users typically identify stiff systems by the fact that numerical differential equation solvers such as `OdeRungeKutta` are inefficient, or else completely fail. Special methods are often required. The most common inefficiency is that a large number of evaluations of \( f(t, y) \) (and hence an excessive amount of computer time) are required to satisfy the accuracy and stability requirements of the software.

---

**OdeRungeKutta Class**

Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method.

```csharp
public class Imsl.Math.OdeRungeKutta
```

**Properties**

**InitialStepsize**

```csharp
public double InitialStepsize {get; set; }
```

The initial internal step size. A `double` specifying the initial internal step size.

**MaxSteps**

```csharp
public int MaxSteps {get; set; }
```

The maximum number of internal steps allowed. An `int` specifying the maximum number of internal steps allowed. Default value is 500.

**MaximumStepsize**

```csharp
public double MaximumStepsize {get; set; }
```

The maximum internal step size. A `double` specifying the maximum internal step size. Default value is 2.

**MinimumStepsize**

```csharp
public double MinimumStepsize {get; set; }
```

The minimum internal step size. A `double` specifying the minimum internal step size. Default value is 0.
Scale

```csharp
public double Scale {get; set; }
```

The scaling factor. A `double` specifying the scaling factor. Default value is `1.0e0`.

Tolerance

```csharp
public double Tolerance {get; set; }
```

The error tolerance. A `double` specifying the error tolerance. Default value is `1.0e-6`.

**Constructor**

OdeRungeKutta

```csharp
public OdeRungeKutta(Imsl.Math.OdeRungeKutta.IFunction f)
```

Constructs an ODE solver to solve the initial value problem \( \frac{dy}{dx} = f(x, y) \).

- `f`: Implementation of interface `IFunction` that defines the right-hand side function \( f(x, y) \).

**Methods**

Solve

```csharp
public void Solve(double x, double xEnd, double[] y)
```

Integrates the ODE system from \( x \) to \( xEnd \).

- `x`: A `double` specifying the independent variable.
- `xEnd`: A `double` specifying the value of \( x \) at which the solution is desired.
- `y`: On input, `double` array containing the initial values.
  On output, `double` array containing the approximate solution.

`Imsl.Math.MaxNumberStepsAllowedException` is thrown if the number of internal steps exceeds `maxSteps` (default 500)

This can be an indication that the ODE system is stiff. This exception can also be thrown if the error tolerance condition could not be met.

`Imsl.Math.ToleranceTooSmallException` is thrown if the computation does not converge on some step

VNorm

```csharp
virtual double VNorm(double[] v, double[] y, double[] ymax)
```

Returns the norm of a vector.

- `v`: A `double` array containing the vector whose norm is to be computed.
- `y`: A `double` array containing the values of the dependent variable.
- `ymax`: A `double` array containing the maximum \( y \) values computed thus far.

Returns — A `double` scalar value representing the norm of the vector \( v \)
Description

Class OdeRungeKutta finds an approximation to the solution of a system of first-order differential equations of the form $y_0 = f(t, y)$ with given initial data. The routine attempts to keep the global error proportional to a user-specified tolerance. This routine is efficient for nonstiff systems where the derivative evaluations are not expensive.

OdeRungeKutta is based on a code designed by Hull, Enright and Jackson (1976, 1977). It uses Runge-Kutta formulas of order five and six developed by J. H. Verner.

Example: Runge-Kutta-Verner ordinary differential equation solver

An ordinary differential equation problem is solved using a solver which implements the Runge-Kutta-Verner method. The solution at time $t=10$ is printed.

```csharp
using System;
using Imsl.Math;

public class OdeRungeKuttaEx1 : OdeRungeKutta.IFunction
{
    public void F(double t, double[] y, double[] yprime)
    {
        yprime[0] = 2.0 * y[0] * (1 - y[1]);
        yprime[1] = -y[1] * (1 - y[0]);
    }

    public static void Main(String[] args)
    {
        double[] y = new double[] {1, 3};
        OdeRungeKutta q = new OdeRungeKutta(new OdeRungeKuttaEx1());
        int nsteps = 10;
        for (int k = 0; k < nsteps; k++)
        {
            q.Solve(k, k + 1, y);
        }
    }
}
```

Output

```
Result = {3.14434167651608,0.348826598519701}
```
OdeRungeKutta.IFunction Interface

Interface for user supplied function to OdeRungeKutta object.

public interface Imsl.Math.OdeRungeKutta.IFunction

Method

F

abstract public void F(double x, double[] y, double[] yprime)
User supplied function to OdeRungeKutta object. On return, yprime contains the
function value at the given point.

x   A double, the point at which the function is to be evaluated.
y   A double array which contains the dependent variable values.
yprime   A double array which, on return, contains the value of the function at (x,y).
Chapter 6: Transforms

Types

class FFT .......................................................... 94
class ComplexFFT .................................................. 98

Usage Notes

Fast Fourier Transforms

A fast Fourier transform (FFT) is simply a discrete Fourier transform that is computed efficiently. Basically, the straightforward method for computing the Fourier transform takes approximately \( n^2 \) operations where \( n \) is the number of points in the transform, while the FFT (which computes the same values) takes approximately \( n \log n \) operations. The algorithms in this chapter are modeled on the Cooley-Tukey (1965) algorithm. Hence, these functions are most efficient for integers that are highly composite; that is, integers that are a product of small primes.

For the two classes, FFT and ComplexFFT, a single instance can be used to transform multiple sequences of the same length. In this situation, the constructor computes the initial setup once. This may result in substantial computational savings. For more information on the use of these classes consult the documentation under the appropriate class name.

Continuous Versus Discrete Fourier Transform

There is, of course, a close connection between the discrete Fourier transform and the continuous Fourier transform. Recall that the continuous Fourier transform is defined (Brigham 1974) as

\[
\hat{f}(\omega) = (\mathcal{F})(\omega) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i \omega t} dt
\]
We begin by making the following approximation:

\[ \hat{f}(\omega) \approx \int_{-T/2}^{T/2} f(t)e^{-2\pi i \omega t} dt \]

\[ = \int_{0}^{T} f(t - T/2)e^{-2\pi i \omega (t - T/2)} dt \]

\[ = e^{\pi i \omega T} \int_{0}^{T} f(t - T/2)e^{-2\pi i \omega t} dt \]

If we approximate the last integral using the rectangle rule with spacing \( h = T/n \), we have

\[ \hat{f}(\omega) \approx e^{\pi i \omega T} h \sum_{k=0}^{n-1} e^{-2\pi i \omega k h} f(kh - T/2) \]

Finally, setting \( \omega = j/T \) for \( j = 0, \ldots, n - 1 \) yields

\[ \hat{f}(j/T) \approx e^{\pi i j h} \sum_{k=0}^{n-1} e^{-2\pi i j k/n} f(kh - T/2) = (-1)^j \sum_{k=0}^{n-1} e^{-2\pi i j k/n} f_k h \]

where the vector \( f^h = (f(-T/2), \ldots, f((n-1)h - T/2)) \). Thus, after scaling the components by \((-1)^h\), the discrete Fourier transform, as computed in ComplexFFT (with input \( f^h \)) is related to an approximation of the continuous Fourier transform by the above formula.

---

### FFT Class

FFT functions.

**public class Imsl.Math.FFT**

**Constructor**

**FFT**

**public FFT(int n)**

Constructs an FFT object.

**n** A int which specifies the array size that this object can handle.
Methods

Backward

public double[] Backward(double[] coef)
Compute the real periodic sequence from its Fourier coefficients.

coef A double array containing the Fourier coefficients.

Returns — A double array containing the periodic sequence.

Forward

public double[] Forward(double[] seq)
Compute the Fourier coefficients of a real periodic sequence.

seq A double array containing the sequence to be transformed.

Returns — A double array containing the transformed sequence.

Description

Class FFT computes the discrete Fourier transform of a real vector of size n. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when n is a product of small prime factors. If n satisfies this condition, then the computational effort is proportional to n log n.

The Forward method computes the forward transform. If n is even, then the forward transform is

\[ q_{2m-1} = \sum_{k=0}^{n-1} p_k \cos \frac{2\pi km}{n} \quad m = 1, \ldots, n/2 \]

\[ q_{2m-2} = -\sum_{k=0}^{n-1} p_k \sin \frac{2\pi km}{n} \quad m = 1, \ldots, n/2 - 1 \]

\[ q_0 = \sum_{k=0}^{n-1} p_k \]

If n is odd, \( q_m \) is defined as above for \( m \) from 1 to \((n - 1)/2\).

Let \( f \) be a real valued function of time. Suppose we sample \( f \) at \( n \) equally spaced time intervals of length \( \delta \) seconds starting at time \( t_0 \). That is, we have

\[ p_i := f(t_0 + i\Delta) \quad i = 0, 1, \ldots, n - 1 \]
We will assume that \( n \) is odd for the remainder of this discussion. The class FFT treats this sequence as if it were periodic of period \( n \). In particular, it assumes that \( f(t_0) = f(t_0 + n\Delta) \). Hence, the period of the function is assumed to be \( T = n\Delta \). We can invert the above transform for \( p \) as follows:

\[
p_m = \frac{1}{n} \left[ q_0 + 2 \sum_{k=0}^{(n-3)/2} q_{2k+1} \cos \frac{2\pi km}{n} - 2 \sum_{k=0}^{(n-3)/2} q_{2k+2} \sin \frac{2\pi km}{n} \right]
\]

This formula is very revealing. It can be interpreted in the following manner. The coefficients \( q \) produced by FFT determine an interpolating trigonometric polynomial to the data. That is, if we define

\[
g(t) = \frac{1}{n} \left[ q_0 + 2 \sum_{k=0}^{(n-3)/2} q_{2k+1} \cos \frac{2\pi k (t - t_0)}{n\Delta} - 2 \sum_{k=0}^{(n-3)/2} q_{2k+2} \sin \frac{2\pi k (t - t_0)}{n\Delta} \right]
\]

\[
= \frac{1}{n} \left[ q_0 + 2 \sum_{k=0}^{(n-3)/2} q_{2k+1} \cos \frac{2\pi k (t - t_0)}{T} - 2 \sum_{k=0}^{(n-3)/2} q_{2k+2} \sin \frac{2\pi k (t - t_0)}{T} \right]
\]

then we have

\[
f(t_0 + (i - 1) \Delta) = g(t_0 + (i - 1)) \Delta
\]

Now suppose we want to discover the dominant frequencies, forming the vector \( P \) of length \( (n + 1)/2 \) as follows:

\[
P_0 := \left| q_0 \right|
\]

\[
P_k := \sqrt{q_{2k-2}^2 + q_{2k-1}^2} \quad k = 1, 2, \ldots, (n - 1)/2
\]

These numbers correspond to the energy in the spectrum of the signal. In particular, \( P_k \) corresponds to the energy level at frequency

\[
\frac{k}{T} = \frac{k}{n\Delta} \quad k = 0, 1, \ldots, \frac{n - 1}{2}
\]
Furthermore, note that there are only \((n + 1)/2 \approx T/(2\Delta)\) resolvable frequencies when \(n\) observations are taken. This is related to the Nyquist phenomenon, which is induced by discrete sampling of a continuous signal. Similar relations hold for the case when \(n\) is even.

If the **Backward** method is used, then the backward transform is computed. If \(n\) is even, then the backward transform is

\[
q_m = p_0 + (-1)^m p_{n-1} + 2 \sum_{k=0}^{n/2-1} p_{2k+1} \cos \frac{2\pi km}{n} - 2 \sum_{k=0}^{n/2-2} p_{2k+2} \sin \frac{2\pi km}{n}
\]

If \(n\) is odd,

\[
q_m = p_0 + 2 \sum_{k=0}^{(n-3)/2} p_{2k+1} \cos \frac{2\pi km}{n} - 2 \sum_{k=0}^{(n-3)/2} p_{2k+2} \sin \frac{2\pi km}{n}
\]

The backward Fourier transform is the unnormalized inverse of the forward Fourier transform. **FFT** is based on the real FFT in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

**Example: Fast Fourier Transform**

The Fourier coefficients of a periodic sequence are computed. The coefficients are then used to reproduce the periodic sequence.

```csharp
using System;
using Imsl.Math;

public class FFTEx1
{
    public static void Main(String[] args)
    {
        double[] x = new double[]{1, 2, 3, 4, 5, 6, 7, 8};
        FFT fft = new FFT(x.Length);
        double[] y = fft.Forward(x);
        double[] z = fft.Backward(y);
        for (int i = 0; i < x.Length; i++)
        {
            z[i] = z[i] / x.Length;
        }
        new PrintMatrix("x").Print(x);
        new PrintMatrix("y").Print(y);
        new PrintMatrix("z").Print(z);
    }
}
```

---

**Transforms**  
**FFT Class: 97**
ComplexFFT Class

Complex FFT.

public class Imsl.Math.ComplexFFT

Constructor

ComplexFFT
    public ComplexFFT(int n)
    Constructs a complex FFT object.
n  A int which specifies the array size that this object can handle.

Methods

Backward

Compute the complex periodic sequence from its Fourier coefficients.

c_coef  A Complex array of Fourier coefficients.

Returns — A Complex array containing the periodic sequence.

Forward

Compute the Fourier coefficients of a complex periodic sequence.

seq  A Complex array containing the sequence to be transformed.

Returns — A Complex array containing the transformed sequence.

Description

Class ComplexFFT computes the discrete complex Fourier transform of a complex vector of size N. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when N is a product of small prime factors. If N satisfies this condition, then the computational effort is proportional to N \log N. This considerable savings has historically led people to refer to this algorithm as the "fast Fourier transform" or FFT.

Specifically, given an N-vector x, method Forward returns

$$c_m = \sum_{n=0}^{N-1} x_n e^{-2\pi i n m / N}$$

Furthermore, a vector of Euclidean norm S is mapped into a vector of norm

$$\sqrt{N} S$$

Finally, note that we can invert the Fourier transform as follows:

$$x_n = \frac{1}{N} \sum_{j=0}^{N-1} c_m e^{2\pi i j n / N}$$
This formula reveals the fact that, after properly normalizing the Fourier coefficients, one has
the coefficients for a trigonometric interpolating polynomial to the data. An unnormalized
inverse is implemented in Backward. ComplexFFT is based on the complex FFT in FFTPACK.
The package, FFTPACK was developed by Paul Swarztrauber at the National Center for
Atmospheric Research.

Specifically, given an $N$-vector $c$, Backward returns

$$s_m = \sum_{n=0}^{N} c_n e^{2\pi i m n / N}$$

Furthermore, a vector of Euclidean norm $S$ is mapped into a vector of norm

$$\sqrt{NS}$$

Finally, note that we can invert the inverse Fourier transform as follows:

$$c_n = \frac{1}{N} \sum_{m=0}^{N-1} s_m e^{-2\pi i m n / N}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, one has
the coefficients for a trigonometric interpolating polynomial to the data. Backward is based on
the complex inverse FFT in FFTPACK. The package, FFTPACK was developed by Paul
Swarztrauber at the National Center for Atmospheric Research.

**Example: Complex FFT**

The Fourier coefficients of a complex periodic sequence are computed. Then the coefficients are
used to try to reproduce the periodic sequence.

```csharp
using System;
using Imsl.Math;

public class ComplexFFTEx1
{
    public static void Main(String[] args)
    {
        Complex[] x = new Complex[]{
            new Complex(1, 8), new Complex(2, 7), new Complex(3, 6),
            new Complex(4, 5), new Complex(5, 4), new Complex(6, 3),
            new Complex(7, 2), new Complex(8, 1)
        };
        ComplexFFT fft = new ComplexFFT(x.Length);
        Complex[] y = fft.Forward(x);
```
Complex[] z = fft.Backward(y);
for (int i = 0; i < x.Length; i++)
{
    z[i] /= x.Length;
}

new PrintMatrix("x").Print(x);
new PrintMatrix("y").Print(y);
new PrintMatrix("z").Print(z);

Output

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1+8i</td>
<td>36+36i</td>
<td>1+8i</td>
</tr>
<tr>
<td>2</td>
<td>2+7i</td>
<td>-2.34314575050762+5.65685424949238i</td>
<td>2+7i</td>
</tr>
<tr>
<td>3</td>
<td>3+6i</td>
<td>-5.65685424949238+2.34314575050762i</td>
<td>3+6i</td>
</tr>
<tr>
<td>4</td>
<td>4+5i</td>
<td>-8</td>
<td>4+5i</td>
</tr>
<tr>
<td>5</td>
<td>5+4i</td>
<td>-13.65685424949238-5.65685424949238i</td>
<td>5+4i</td>
</tr>
<tr>
<td>6</td>
<td>6+3i</td>
<td>8+1i</td>
<td>6+3i</td>
</tr>
<tr>
<td>7</td>
<td>7+2i</td>
<td>8+1i</td>
<td>7+2i</td>
</tr>
</tbody>
</table>

Chapter 6. Transforms
Chapter 7: Nonlinear Equations

Types

- class ZeroPolynomial ................................................................. 104
- class ZeroFunction ................................................................. 108
- interface ZeroFunction.IFunction ........................................... 111
- class ZeroSystem ................................................................. 111
- interface ZeroSystem.IFunction .............................................. 113
- interface ZeroSystem.IJacobian ............................................. 114

Usage Notes

Zeros of a Polynomial

A polynomial function of degree \( n \) can be expressed as follows:

\[
p(z) = a_n z^n + a_{n-1} z^{n-1} + \cdots + a_1 z + a_0
\]

where \( a_n \neq 0 \). The ZeroPolynomial class finds zeros of a polynomial with real or complex coefficients using Aberth’s method.

Zeros of a Function

The ZeroFunction class uses Muller’s method to find the real zeros of a real-valued function.

Root of System of Equations

A system of equations can be stated as follows:

\[
f_i(x) = 0, \text{ for } i = 1, 2, \ldots, n
\]
where \( x \in \mathbb{R}^n \), and \( f_i : \mathbb{R}^n \rightarrow \mathbb{R} \). The \texttt{ZeroSystem} class uses a modified hybrid method due to M.J.D. Powell to find the zero of a system of nonlinear equations.

---

### ZeroPolynomial Class

The \texttt{ZeroPolynomial} class computes the zeros of a polynomial with complex coefficients, Aberth’s method.

```csharp
public class Imsl.Math.ZeroPolynomial

// Property

MaximumIterations
public int MaximumIterations { get; set; }
    The maximum number of iterations allowed. An \texttt{int} which specifies the maximum number of iterations allowed. The default value is 30.

// Constructor

ZeroPolynomial
public ZeroPolynomial()
    Creates an instance of the solver.

// Methods

ComputeRoots
public Imsl.Math.Complex[] ComputeRoots(double[] coef)
    Computes the roots of the polynomial with real coefficients.
    \[ p(x) = \text{coef}[n] \times x^n + \text{coef}[n-1] \times x^{n-1} + \ldots + \text{coef}[0] \]
    \texttt{coef}  A \texttt{double} array containing the polynomial coefficients.
    Returns — A \texttt{Complex} array containing the roots of the polynomial.
    \texttt{Imsl.Math.DidNotConvergeException} is thrown if the iteration did not converge.

ComputeRoots
    Computes the roots of the polynomial with Complex coefficients.
```
\[ p(x) = \text{coef}[n] \times x^n + \text{coef}[n-1] \times x^{n-1} + \ldots + \text{coef}[0] \]

**coef**  A Complex array containing the polynomial coefficients.

Returns — A Complex array containing the roots of the polynomial.

**Imsl.Math.DidNotConvergeException** is thrown if if the iteration for the zeros did not converge.

**GetRadius**

```java
public double GetRadius(int index)
```

Returns an a-posteriori absolute error bound on the root.

NaN is returned if the corresponding root cannot be represented as floating point due to overflow or underflow or if the roots have not yet been computed.

**index**  An int specifying the (0-based) index of the root whose error bound is to be returned.

Returns — A double representing the error bound on the index-th root.

**GetRoot**

```java
public Imsl.Math.Complex GetRoot(int index)
```

Returns a zero of the polynomial.

**index**  An int which specifies the (0-based) index of the root to be returned.

Returns — A Complex which represents the index-th root of the polynomial.

**GetRoots**

```java
public Imsl.Math.Complex[] GetRoots()
```

Returns the zeros of the polynomial.

Returns — A Complex array containing the roots of the polynomial.

**GetStatus**

```java
public bool GetStatus(int index)
```

Returns the error status of a root.

It is **false** if the approximation of the index-th root has been carried out successfully, for example, the computed approximation can be viewed as the exact root of a slightly perturbed polynomial. It is true if more iterations are needed for the index-th root.

**index**  An int representing the (0-based) index of the root whose error status is to be returned.

Returns — A boolean representing the error status on the index-th root.
Description

This class is a translation of a Fortran code written by Dario Andrea Bini, University of Pisa, Italy (bini@dm.unipi.it). Numerical computation of polynomial zeros by means of Aberth’s method, Numerical Algorithms, 13 (1996), pp. 179-200.

The original Fortran code includes the following notice.

All the software contained in this library is protected by copyright Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.


Example 1: Zeros of a Polynomial

The zeros of a polynomial with real coefficients are computed.

```csharp
using System;
using Imsl.Math;

public class ZeroPolynomialEx1
{
    public static void Main(String[] args)
    {
        double[] coef = new double[]{- 2, 4, - 3, 1};

        ZeroPolynomial zp = new ZeroPolynomial();
        Complex[] root = zp.ComputeRoots(coef);

        for (int k = 0; k < root.Length; k++)
        {
            Console.Out.WriteLine("root = " + root[k]);
            Console.Out.WriteLine(" radius = " + zp.GetRadius(k));
            Console.Out.WriteLine(" status = " + zp.GetStatus(k));
        }
    }
}
```

```
Example 2: Zeros of a Polynomial with Complex Coefficients

The zeros of a polynomial with Complex coefficients are computed.

```csharp
class ZeroPolynomialEx2
{
    public static void Main(String[] args)
    {
        // Find zeros of \(z^3-(3+6i)z^2+(-8+12i)z+10\)
        Complex[] coef = new Complex[]{
            new Complex(10),
            new Complex(-8, 12),
            new Complex(-3, -6),
            new Complex(1)};

        ZeroPolynomial zp = new ZeroPolynomial();
        Complex[] root = zp.ComputeRoots(coef);

        for (int k = 0; k < root.Length; k++)
        {
            Console.Out.WriteLine("root = \" + root[k].ToString() + \"");
            Console.Out.WriteLine("radius = \" + zp.GetRadius(k).ToString("0.00e+0\") + \"");
        }
    }
}
```

Output

```
root = 1+1i
radius = 6.11e-14
status = False
root = 0.999999999999999+2i
radius = 1.95e-13
status = False
root = 1+3i
```
ZeroFunction Class

The ZeroFunction class uses Muller’s method to find the zeros of a univariate function, f(x).

public class Imsl.Math.ZeroFunction

Properties

AbsoluteError

public double AbsoluteError {get; set; }
The first stopping criterion. A double value specifying the first stopping criterion.
A zero \( x[i] \) is accepted if \( |f(x[i])| \) is less than this tolerance. Its default value is about 1.0e-8.

MaximumIterations

public int MaximumIterations {get; set; }
The maximum number of iterations allowed per root. An int specifying the maximum number of iterations allowed per root.
The default value is 100.

RelativeError

public double RelativeError {get; set; }
The second stopping criterion is the relative error. A double value specifying the second stopping criterion.
A zero \( x[i] \) is accepted if the relative change of two successive approximations to \( x[i] \) is less than this tolerance. Its default value is about 1.0e-8.

Spread

public double Spread {get; set; }
The spread. The new spread.
The default value is 1.0.

SpreadTolerance

public double SpreadTolerance {get; set; }
The spread criteria for multiple zeros. A double value specifying the spread tolerance.
If the zero \( x[i] \) has been computed and \( |x[i] - x[j]| < SpreadTolerance \), where \( x[j] \) is a previously computed zero, then the computation is restarted with a guess equal to \( x[i] + Spread \). The default value for SpreadTolerance is 1.0e-5.
Constructor

ZeroFunction
    public ZeroFunction()
    Creates an instance of the solver.

Methods

AllConverged
    public bool AllConverged()
    Returns true if the iterations for all of the roots have converged.
    Returns — A boolean value specifying whether the roots have converged.

ComputeZeros
    public double[] ComputeZeros(Imsl.Math.ZeroFunction.IFunction f, double[] guess)
    Returns the zeros of a univariate function.
    f    The ZeroFunction.IFunction to be integrated.
    guess A double array containing an initial guess of the zeros. A zero will be found for each point in guess.
    Returns — A double array containing the zero of the univariate function.

GetIterations
    public int GetIterations(int nRoot)
    Returns the number of iterations used to compute a root.
    nRoot An int specifying the index of the root.
    Returns — An int specifying the number of iterations necessary to compute a root.

Description

ZeroFunction computes n real zeros of a real function f. Given a user-supplied function f(x) and an n-vector of initial guesses x₁, x₂, ..., xₙ, the routine uses Muller’s method to locate n real zeros of f, that is, n real values of x for which f(x) = 0. The routine has two convergence criteria. The first requires the absolute value of the function be less than the AbsoluteError. The second requires that the relative change of any two successive approximations to an xᵢ be less than RelativeError. Here, xᵢᵐ is the m-th approximation to xᵢ. Let AbsoluteError be ε₁, and RelativeError be ε₂. The criteria may be stated mathematically as follows:

Criterion 1:

\[ |f(xᵢᵐ)| < ε₁ \]
Criterion 2:

\[
\left| \frac{x_{i+1}^m - x_i^m}{x_i^m} \right| < \varepsilon_2
\]

"Convergence" is the satisfaction of either criterion.

**Example: Zeros of a Univariate Function**

In this example 3 zeros of the sin function are found.

```csharp
using System;
using Imsl.Math;

public class ZeroFunctionEx1 : ZeroFunction.IFunction
{
    public double F(double x)
    {
        return Math.Sin(x);
    }

    public static void Main(String[] args)
    {
        ZeroFunction.IFunction fcn = new ZeroFunctionEx1();

        ZeroFunction zf = new ZeroFunction();
        double[] guess = new double[]{5, 18, -6};
        double[] zeros = zf.ComputeZeros(fcn, guess);
        for (int k = 0; k < zeros.Length; k++)
        {
            Console.Out.WriteLine(zeros[k] + " = " + (zeros[k] / Math.PI) + " pi"u);
        }
    }
}
```

**Output**

6.28318530717956 = 1.99999999999999 pi
18.8495559215629 = 6.0000000000077 pi
-6.28318530717964 = -2.00000000000002 pi
ZeroFunction.IFunction Interface

Interface for the user supplied function to ZeroFunction.

```java
public interface Imsl.Math.ZeroFunction.IFunction
```

**Method**

```java
abstract public double F(double x)
```

The user supplied function to ZeroFunction.

Returns the value of the function at the given point.

- **x** A `double` specifying the point at which the function is to be evaluated.

Returns — A `double` specifying the value of the function at x.

ZeroSystem Class

Solves a system of n nonlinear equations f(x) = 0 using a modified Powell hybrid algorithm.

```java
public class Imsl.Math.ZeroSystem
```

**Properties**

```java
public int MaximumIterations {get; set; }
```

The maximum number of iterations allowed. An `int` specifying the maximum number of iterations allowed.

The default value is 200.

```java
public double RelativeError {get; set; }
```

The relative error tolerance. A `double` specifying the relative error tolerance.

The root is accepted if the relative error between two successive approximations to this root is within `errorRelative`. The default is the square root of the precision, about 1.0e-08.
**Constructor**

ZeroSystem

```java
public ZeroSystem(int n)
```

Creates an object to find the zeros of a system of n equations.

- **n**  The number of equations that the solver handles.

**Methods**

**SetGuess**

```java
public void SetGuess(double[] guess)
```

Sets initial guess for the the solution.

- **guess**  A double array containing the initial guess.

**Solve**

```java
public double[] Solve(Imsl.Math.ZeroSystem.IFunction f)
```

Solve a system of nonlinear equations using the Levenberg-Marquardt algorithm.

See Also: Imsl.Math.ZeroSystem.IJacobian (p. 114)

- **f**  Defines a ZeroSystem.IFunction whose zero is to be found. If f implements a ZeroSystem.IJacobian then its Jacobian is used. Otherwise finite difference is used.

Returns — A double array containing the solution.

- Imsl.Math.TooManyIterationsException is thrown if the maximum number of iterations is exceeded
- Imsl.Math.ToleranceTooSmallException is thrown if the error tolerance is too small
- Imsl.Math.DidNotConvergeException is thrown if the algorithm does not converge

**Description**

ZeroSystem is based on the MINPACK subroutine HYBRD1, which uses a modification of M.J.D. Powell's hybrid algorithm. This algorithm is a variation of Newton's method, which uses a finite-difference approximation to the Jacobian and takes precautions to avoid large step sizes or increasing residuals. For further description, see More et al. (1980).

A finite-difference method is used to estimate the Jacobian. Whenever the exact Jacobian can be easily provided, f should implement ZeroSystem.IJacobian.

**Example: Solve a System of Nonlinear Equations**

A system of nonlinear equations is solved.
using System;
using Imsl.Math;

public class ZeroSystemEx1 : ZeroSystem.IFunction
{
    public void F(double[] x, double[] f)
    {
    }

    public static void Main(String[] args)
    {
        ZeroSystem zf = new ZeroSystem(3);
        zf.SetGuess(new double[]{4, 4, 4});
        new PrintMatrix("zeros").Print(zf.Solve(new ZeroSystemEx1()));
    }
}

Output

    zeros
    0
    0 0.99999999995498
    1 2.00000000000656
    2 2.99999999999468

ZeroSystem.IFunction Interface

Public interface for user supplied function to ZeroSystem object.

public interface Imsl.Math.ZeroSystem.IFunction

Method

F

abstract public void F(double[] x, double[] fvalue)
    On return, fvalue contains the function value at the given point.

    x A double array which contains the point at which the function is to be evaluated.
    The contents of this array must not be altered by this function.
fvalue  A double array which, on return, contains the value of the function at x.

ZeroSystem.IJacobian Interface

Public interface for user supplied function to ZeroSystem object.

public interface Imsl.Math.ZeroSystem.IJacobian implements Method

Jacobian

abstract public void Jacobian(double[] x, double[,] jac)
On return, jac contains the value of the Jacobian at the given point.

x  A double array which contains the point at which the Jacobian is to be evaluated.
The contents of this array must not be altered by this function.

jac  A double matrix which, on return, contains the value of the Jacobian at x. The
value of jac[i,j] is the derivative of f[i] with respect to x[j].
Chapter 8: Optimization

Types

class MinUncon .............................................................. 117
interface MinUncon.IFunction ........................................ 122
interface MinUncon.IDerivative ....................................... 122
class MinUnconMultiVar .................................................. 122
interface MinUnconMultiVar.IFunction .............................. 128
interface MinUnconMultiVar.IGradient .............................. 128
class NonlinLeastSquares ............................................... 129
interface NonlinLeastSquares.IFunction ............................ 135
interface NonlinLeastSquares.IJacobian ............................ 135
class LinearProgramming ............................................... 135
class QuadraticProgramming .......................................... 140
class MinConGenLin ....................................................... 144
interface MinConGenLin.IFunction ................................... 151
interface MinConGenLin.IGradient ................................... 151
class BoundedLeastSquares ............................................. 152
interface BoundedLeastSquares.IFunction .......................... 158
interface BoundedLeastSquares.IJacobian ........................... 159
class MinConNLP .......................................................... 159
interface MinConNLP.IFunction ....................................... 168
interface MinConNLP.IGradient ....................................... 168

Usage Notes

Unconstrained Minimization

The unconstrained minimization problem can be stated as follows:

$$\min_{x \in \mathbb{R}^n} f(x)$$
where $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is continuous and has derivatives of all orders required by the algorithms. The functions for unconstrained minimization are grouped into three categories: univariate functions, multivariate functions, and nonlinear least-squares functions.

For the univariate functions, it is assumed that the function is unimodal within the specified interval. For discussion on unimodality, see Brent (1973).

The class `MinUnconMultiVar` finds the minimum of a multivariate function using a quasi-Newton method. The default is to use a finite-difference approximation of the gradient of $f(x)$. Here, the gradient is defined to be the vector

$$
\nabla f(x) = \left[ \frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, ..., \frac{\partial f(x)}{\partial x_n} \right]
$$

However, when the exact gradient can be easily provided, the gradient should be provided by implementing the interface `MinUnconMultiVar.Gradient`.

The nonlinear least-squares function uses a modified Levenberg-Marquardt algorithm. The most common application of the function is the nonlinear data-fitting problem where the user is trying to fit the data with a nonlinear model.

These functions are designed to find only a local minimum point. However, a function may have many local minima. Try different initial points and intervals to obtain a better local solution.

**Linearly Constrained Minimization**

The linearly constrained minimization problem can be stated as follows:

$$
\min_{x \in \mathbb{R}^n} f(x)
$$

subject to $A_1x = b_1$

where $f : \mathbb{R}^n \rightarrow \mathbb{R}$, $A_1$ is a coefficient matrix, and $b_1$ is a vector. If $f(x)$ is linear, then the problem is a linear programming problem. If $f(x)$ is quadratic, the problem is a quadratic programming problem.

The class `LinearProgramming` uses a revised simplex method to solve small- to medium-sized linear programming problems. No sparsity is assumed since the coefficients are stored in full matrix form.

The class `QuadraticProgramming` is designed to solve convex quadratic programming problems using a dual quadratic programming algorithm. If the given Hessian is not positive definite, then `QuadraticProgramming` modifies it to be positive definite. In this case, output should be interpreted with care because the problem has been changed slightly. Here, the Hessian of $f(x)$ is defined to be the $n \times n$ matrix

$$
\nabla^2 f(x) = \left[ \frac{\partial^2 f(x)}{\partial x_i \partial x_j} \right]
$$
Nonlinearly Constrained Minimization

The nonlinearly constrained minimization problem can be stated as follows:

$$\min_{x \in \mathbb{R}^n} f(x)$$
subject to $g_i(x) = 0$ for $i = 1, 2, \ldots, m_1$
$$g_i(x) \geq 0 \text{ for } i = m_1 + 1, \ldots, m$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $g_i : \mathbb{R}^n \to \mathbb{R}$, for $i = 1, 2, \ldots, m$.

The class MinConNLP uses a sequential equality constrained quadratic programming algorithm to solve this problem. A more complete discussion of this algorithm can be found in the documentation.

MinUncon Class

Finds the minimum point for a smooth univariate function using function and optionally first derivative evaluations.

```java
public class Imsl.Math.MinUncon
```

Properties

**Accuracy**

```java
public double Accuracy {get; set; }
```

The required absolute accuracy in the final value returned by the ComputeMin method. A `double` scalar value specifying the required absolute accuracy in the final value returned by the ComputeMin method.

By default, the required accuracy is set to 1.0e-8.

**Bound**

```java
public double Bound {get; set; }
```

The amount by which X may be changed from its initial value, Guess. A `double` scalar value specifying the amount by which X may be changed from its initial value. In other words, X is restricted to the interval $[\text{Guess-Bound}, \text{Guess+Bound}]$.

By default, `Bound` is set to 100.

**DerivTolerance**

```java
public double DerivTolerance {get; set; }
```

The derivative tolerance used by member method ComputeMin to decide if the current point is a local minimum. A `double` scalar value specifying the derivative tolerance used by member method ComputeMin.
This is the second stopping criterion. $x$ is returned as a solution when $G(x)$ is less than or equal to `DerivTolerance`. `DerivTolerance` should be nonnegative, otherwise zero will be used. By default, `DerivTolerance` is set to 1.0e-8.

**Guess**

```csharp
public double Guess {get; set; }
```

The initial guess of the minimum point of the input function. A `double` scalar value specifying the initial guess of the minimum point of the input function.

By default, an initial guess of 0.0 is used.

**Step**

```csharp
public double Step {get; set; }
```

The stepsize to use when changing $x$. A `double` scalar value specifying the order of magnitude estimate of the required change in $x$ when stepping towards the minimum.

By default, `Step` is set to 0.1.

### Constructor

**MinUncon**

```csharp
public MinUncon()
```

Unconstrained minimum constructor for a smooth function of a single variable of type `double`.

### Method

**ComputeMin**

```csharp
public double ComputeMin(Imsl.Math.MinUncon.IFunction f)
```

Return the minimum of a smooth function of a single variable of type `double` using function values only or using function values and derivatives.

- `f` The `MinUncon.IFunction` whose minimum is to be found. An attempt to find the minimum is made using function values only.

Returns — A `double` scalar value containing the minimum of the input function.

### Description

`MinUncon` uses two separate algorithms to compute the minimum depending on what the user supplies as the function `f`.

If `f` defines the function whose minimum is to be found `MinUncon` uses a safeguarded quadratic interpolation method to find a minimum point of a univariate function. Both the code and the underlying algorithm are based on the routine `ZXLSF` written by M.J.D. Powell at the University of Cambridge.
MinUncon finds the least value of a univariate function, \( f \), where \( f \) is a MinUncon.IFunction. Optional data include an initial estimate of the solution, and a positive number specified by the Bound property. Let \( x_0 = \text{Guess} \) where \( \text{Guess} \) is specified by the Guess property and \( b = \text{Bound} \). Usually, the algorithm begins the search by moving from \( x_0 \) to \( x = x_0 + s \), where \( s = \text{Step} \). \( \text{Step} \) is set by the Step property. If \( \text{Step} \) is not called then \( \text{Step} \) is set to 0.1. \( \text{Step} \) may be positive or negative. The first two function evaluations indicate the direction to the minimum point, and the search strides out along this direction until a bracket on a minimum point is found or until \( x \) reaches one of the bounds \( x_0 \pm b \). During this stage, the step length increases by a factor of between two and nine per function evaluation; the factor depends on the position of the minimum point that is predicted by quadratic interpolation of the three most recent function values.

When an interval containing a solution has been found, we will have three points, \( x_1, x_2, \) and \( x_3 \), with \( x_1 < x_2 < x_3 \) and \( f(x_2) \leq f(x_1) \) and \( f(x_2) \leq f(x_3) \). There are three main ingredients in the technique for choosing the new \( x \) from these three points. They are (i) the estimate of the minimum point that is given by quadratic interpolation of the three function values, (ii) a tolerance parameter \( \varepsilon \), that depends on the closeness of \( f \) to a quadratic, and (iii) whether \( x_2 \) is near the center of the range between \( x_1 \) and \( x_3 \) or is relatively close to an end of this range. In outline, the new value of \( x \) is as near as possible to the predicted minimum point, subject to being at least \( \varepsilon \) from \( x_2 \), and subject to being in the longer interval between \( x_1 \) and \( x_3 \) or \( x_2 \) and \( x_3 \) when \( x_2 \) is particularly close to \( x_1 \) or \( x_3 \). There is some elaboration, however, when the distance between these points is close to the required accuracy; when the distance is close to the machine precision; or when \( \varepsilon \) is relatively large.

The algorithm is intended to provide fast convergence when \( f \) has a positive and continuous second derivative at the minimum and to avoid gross inefficiencies in pathological cases, such as

\[
f(x) = x + 1.001 |x|
\]

The algorithm can make \( \varepsilon \) large automatically in the pathological cases. In this case, it is usual for a new value of \( x \) to be at the midpoint of the longer interval that is adjacent to the least calculated function value. The midpoint strategy is used frequently when changes to \( f \) are dominated by computer rounding errors, which will almost certainly happen if the user requests an accuracy that is less than the square root of the machine precision. In such cases, the routine claims to have achieved the required accuracy if it knows that there is a local minimum point within distance \( \delta \) of \( x \), where \( \delta = xacc \), specified by the Accuracy property even though the rounding errors in \( f \) may cause the existence of other local minimum points nearby. This difficulty is inevitable in minimization routines that use only function values, so high precision arithmetic is recommended.

If \( f \) is a MinUncon.IDerivative then MinUncon uses a descent method with either the secant method or cubic interpolation to find a minimum point of a univariate function. It starts with an initial guess and two endpoints. If any of the three points is a local minimum point and has least function value, the routine terminates with a solution. Otherwise, the point with least function value will be used as the starting point.

From the starting point, say \( x_c \), the function value \( f_c = f(x_c) \), the derivative value \( g_c = g(x_c) \),
and a new point \( x_n \) defined by \( x_n = x_c - g_c \) are computed. The function \( f_n = f(x_n) \), and the derivative \( g_n = g(x_n) \) are then evaluated. If either \( f_n \geq f_c \) or \( g_n \) has the opposite sign of \( g_c \), then there exists a minimum point between \( x_c \) and \( x_n \); and an initial interval is obtained. Otherwise, since \( x_c \) is kept as the point that has lowest function value, an interchange between \( x_n \) and \( x_c \) is performed. The secant method is then used to get a new point

\[
x_s = x_c - g_c \left( \frac{g_n - g_c}{x_n - x_c} \right)
\]

Let \( x_n \leftarrow x_s \) and repeat this process until an interval containing a minimum is found or one of the convergence criteria is satisfied. The convergence criteria are as follows:

Criterion 1:

\[
|x_c - x_n| \leq \varepsilon_c
\]

Criterion 2:

\[
|g_c| \leq \varepsilon_g
\]

where \( \varepsilon_c = \max\{1.0, |x_c|\} \varepsilon \), \( \varepsilon \) is a relative error tolerance and \( \varepsilon_c \) is a gradient tolerance.

When convergence is not achieved, a cubic interpolation is performed to obtain a new point. The function and derivative are then evaluated at that point; and accordingly, a smaller interval that contains a minimum point is chosen. A safeguarded method is used to ensure that the interval reduces by at least a fraction of the previous interval. Another cubic interpolation is then performed, and this procedure is repeated until one of the stopping criteria is met.

**Example 1: Minimum of a smooth function**

The minimum of \( e^x - 5x \) is found using function evaluations only.

```csharp
using System;
using Imsl.Math;

class MinUnconEx1 : MinUncon.IFunction
{
    public double F(double x)
    {
        return Math.Exp(x) - 5.0 * x;
    }
}

class Main(String[] args)
{
...
```
MinUncon zf = new MinUncon();
zf.Guess = 0.0;
zf.Accuracy = 0.001;
MinUncon.IFunction fcn = new MinUnconEx1();
Console.Out.WriteLine("Minimum is " + zf.ComputeMin(fcn));
}

Output

Minimum is 1.60941759992003

Example 2: Minimum of a smooth function

The minimum of $e^x - 5x$ is found using function evaluations and first derivative evaluations.

using System;
using Imsl.Math;

class MinUnconEx2 : MinUncon.IDerivative
{
    public double F(double x)
    {
        return Math.Exp(x) - 5.0 * x;
    }

    public double Derivative(double x)
    {
        return Math.Exp(x) - 5.0;
    }

    public static void Main(String[] args)
    {
        MinUncon zf = new MinUncon();
zf.Guess = 0.0;
zf.Accuracy = .001;
double x = zf.ComputeMin(new MinUnconEx2());
Console.Out.WriteLine("x = " + x);
    }

Output

x = 1.61001131622703
**MinUncon.IFunction Interface**

Interface for the user supplied function for the smooth function of a single variable to be minimized.

```csharp
public interface Imsl.Math.MinUncon.IFunction
```

**Method**

```csharp
abstract public double F(double x)
```

Smooth function of a single variable to be minimized.

- `x`: A `double`, the point at which the function is to be evaluated.
- Returns — A `double`, the value of the function at `x`.

**MinUncon.IDerivative Interface**

Interface for the smooth function of a single variable to be minimized and its derivative.

```csharp
public interface Imsl.Math.MinUncon.IDerivative implements MinUncon.IFunction
```

**Method**

```csharp
abstract public double Derivative(double x)
```

Derivative of the smooth function of a single variable to be minimized.

- `x`: A `double`, the point at which the derivative of the function is to be evaluated.
- Returns — A `double`, the value of the derivative of the function at `x`.

**MinUnconMultiVar Class**

Minimizes a multivariate function using a quasi-Newton method.

```csharp
public class Imsl.Math.MinUnconMultiVar
```
Properties

Digits

public double Digits {get; set; }

The number of good digits in the function. A double scalar value specifying the number of good digits in the user supplied function.

By default, Digits is set to 15.75.

ErrorStatus

public int ErrorStatus {get; }

The non-fatal error status. An int specifying the non-fatal error status:

<table>
<thead>
<tr>
<th>ErrorStatus</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The last global step failed to locate a lower point than the current x value. The current x may be an approximate local minimizer and no more accuracy is possible or the step tolerance may be too large.</td>
</tr>
<tr>
<td>2</td>
<td>Relative function convergence; both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.</td>
</tr>
<tr>
<td>3</td>
<td>Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or the step tolerance is too big.</td>
</tr>
</tbody>
</table>

FalseConvergenceTolerance

public double FalseConvergenceTolerance {get; set; }

The false convergence tolerance. A double scalar value specifying the false convergence tolerance.

By default, 2.22044604925031308e-14 is used as the false convergence tolerance.

Fscale

public double Fscale {get; set; }

The function scaling value for scaling the gradient. A double scalar specifying the function scaling value for scaling the gradient.

By default, the value of this scalar is set to 1.0.

GradientTolerance

public double GradientTolerance {get; set; }

The gradient tolerance used to compute the gradient. A double specifying the gradient tolerance used to compute the gradient.

By default, the cube root of machine precision squared is used to compute the gradient.

Ihess

public int Ihess {get; set; }
The Hessian initialization parameter. An int scalar value specifying the Hessian initialization parameter.

By default, Ihess is set to 0.0 and the Hessian is initialized to the identity matrix. If this member function is called and Ihess is set to anything other than 0.0, the Hessian is initialized to the diagonal matrix containing

\[
\max(\text{abs}(f(x_{\text{guess}})), fscale) \times xscale \times xscale
\]

where xguess is the initial guess of the computed solution and xscale is the scaling vector for the variables.

**Iterations**

```
public int Iterations {get; }
```

The number of iterations used to compute a minimum. An int specifying the number of iterations used to compute the minimum.

**MaxIterations**

```
public int MaxIterations {get; set; }
```

The maximum number of iterations allowed. An int specifying the maximum number of iterations allowed.

By default, the maximum number of iterations is set to 100.

**MaximumStepsize**

```
public double MaximumStepsize {get; set; }
```

The maximum allowable stepsize to use. A nonnegative double value specifying the maximum allowable stepsize.

By default, maximum stepsize is set to a value based on a scaled Guess.

**RelativeTolerance**

```
public double RelativeTolerance {get; set; }
```

The relative function tolerance. A double scalar value specifying the relative function tolerance.

By default, 3.66685e-11 is used as the relative function tolerance.

**StepTolerance**

```
public double StepTolerance {get; set; }
```

The scaled step tolerance to use when changing x. A double scalar value specifying the scaled step tolerance.

The i-th component of the scaled step between two points x and y is computed as

\[
\text{abs}(x(i) - y(i)) / \max(\text{abs}(x(i)), 1/xscale(i))
\]

where xscale is the scaling vector for the variables.

By default, the scaled step tolerance is set to 3.66685e-11.

**Constructor**

MinUnconMultiVar
public MinUnconMultiVar(int n)
Unconstrained minimum constructor for a function of n variables of type double.

n  An int scalar value which defines the number of variables of the function whose minimum is to be found.

Methods

ComputeMin
public double[] ComputeMin(Imsl.Math.MinUnconMultiVar.IFunction f)
Return the minimum point of a function of n variables of type double using a finite-difference gradient or using a user-supplied gradient.

f can be used to supply a gradient of the function. If f implements IGradient then the user-supplied gradient is used. Otherwise, an attempt to find the minimum is made using a finite-difference gradient.

f  The MinUnconMultiVar.IFunction whose minimum is to be found.

Returns — A double array containing the point at which the minimum of the input function occurs.

Imsl.Math.FalseConvergenceException is thrown if the iterates appear to be converging to a noncritical point

Imsl.Math.MaxIterationsException is thrown if the maximum number of iterations is exceeded

Imsl.Math.UnboundedBelowException is thrown if five consecutive steps of the maximum allowable stepsize have been taken, either the function is unbounded below, or has a finite asymptote in some direction or the maximum allowable step size is too small

SetGuess
public void SetGuess(double[] guess)
Sets the initial guess of the minimum point of the input function.

By default, the elements of this array are set to 0.0.

guess  A double array specifying the initial guess of the minimum point of the input function.

SetXscale
public void SetXscale(double[] xscale)
Sets the diagonal scaling matrix for the variables.

By default, the elements of this array are set to 1.0.

xscale  A double array specifying the diagonal scaling matrix for the variables.

System.ArgumentException is thrown if any of the elements of Xscale is less than or equal to or equal to 0
Description

Class MinUnconMultivar uses a quasi-Newton method to find the minimum of a function $f(x)$ of $n$ variables. The problem is stated as follows:

$$\min_{x \in \mathbb{R}^n} f(x)$$

Given a starting point $x_c$, the search direction is computed according to the formula

$$d = -B^{-1}g_c$$

where $B$ is a positive definite approximation of the Hessian, and $g_c$ is the gradient evaluated at $x_c$. A line search is then used to find a new point

$$x_n = x_c + \lambda d, \lambda > 0$$

such that

$$f(x_n) \leq f(x_c) + \alpha g^T d, \alpha \in (0, 0.5)$$

Finally, the optimality condition $||g(x)|| \leq \varepsilon$ where $\varepsilon$ is a gradient tolerance.

When optimality is not achieved, $B$ is updated according to the BFGS formula

$$B \leftarrow B - \frac{Bss^TB}{s^T B s} + \frac{yy^T}{y^T s}$$

where $s = x_n - x_c$ and $y = g_n - g_c$. Another search direction is then computed to begin the next iteration. For more details, see Dennis and Schnabel (1983, Appendix A).

In this implementation, the first stopping criterion for MinUnconMultivar occurs when the norm of the gradient is less than the given gradient tolerance property, GradientTolerance. The second stopping criterion for MinUnconMultivar occurs when the scaled distance between the last two steps is less than the step tolerance property, StepTolerance.

Since by default, a finite-difference method is used to estimate the gradient. An inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. Supply the gradient for a more accurate gradient evaluation (MinConMultiVar.IGradient).
Example 1: Minimum of a multivariate function

The minimum of $100(x_2 - x_1^2)^2 + (1 - x_1)^2$ is found using function evaluations only.

```csharp
using System;
using Imsl.Math;

public class MinUnconMultiVarEx1 : MinUnconMultiVar.IFunction
{
    public double F(double[] x)
    {
        return 100.0 * ((x[1] - x[0] * x[0]) * (x[1] - x[0] * x[0])) +
               (1.0 - x[0]) * (1.0 - x[0]);
    }

    public static void Main(String[] args)
    {
        MinUnconMultiVar solver = new MinUnconMultiVar(2);
        solver.SetGuess(new double[]{ -1.2, 1.0 });
        double[] x = solver.ComputeMin(new MinUnconMultiVarEx1());
        Console.Out.WriteLine
            ("Minimum point is "+x[0]+", "+x[1]+")
    }
}
```

Output

Minimum point is (0.9999996726613, 0.99999993304521)

Example 2: Minimum of a multivariate function

The minimum of $100(x_2 - x_1^2)^2 + (1 - x_1)^2$ is found using function evaluations and a user supplied gradient.

```csharp
using System;
using Imsl.Math;

public class MinUnconMultiVarEx2 : MinUnconMultiVar.IGradient
{
    public double F(double[] x)
    {
        return 100.0 * ((x[1] - x[0] * x[0]) * (x[1] - x[0] * x[0])) +
               (1.0 - x[0]) * (1.0 - x[0]);
    }

    public void Gradient(double[] x, double[] gp)
    {
        gp[0] = -400.0 * (x[1] - x[0] * x[0]) * x[0] - 2.0 * (1.0 - x[0]);
    }
}
```
\[
gp[1] = 200.0 \times (x[1] - x[0] \times x[0]);
\]

```csharp
public static void Main(String[] args)
{
    MinUnconMultiVar solver = new MinUnconMultiVar(2);
    solver.SetGuess(new double[]{1.2, 1.0});
    double[] x = solver.ComputeMin(new MinUnconMultiVarEx2());
    Console.WriteLine
        ("Minimum point is (" + x[0] + ", " + x[1] + ")");
}
```

**Output**

Minimum point is (0.999999966882301, 0.999999932254245)

---

**MinUnconMultiVar.IFunction Interface**

Interface for the user supplied multivariate function to be minimized.

```csharp
public interface Imsl.Math.MinUnconMultiVar.IFunction
```

**Method**

```csharp
abstract public double F(double[] x)
```

Multivariate function to be minimized.

- **x** A double array, the point at which the function is to be evaluated.
- **Returns** — A double, the value of the function at x.

---

**MinUnconMultiVar.IGradient Interface**

Interface for the user supplied multivariate function to be minimized and its gradient.

```csharp
public interface Imsl.Math.MinUnconMultiVar.IGradient implements
```

---

**IMSL C# Numerical Library**

128 • MinUnconMultiVar.IFunction Interface
Method

Gradient

abstract public void Gradient(double[] x, double[] gvalue)
On return, gvalue contains the value of the gradient, of the function, at x.

x  A double array, the point at which the gradient of the function is to be evaluated.
gvalue A double array which, on return, contains the value of the gradient, of the function, at x.

NonlinLeastSquares Class

Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm.

public class Imsl.Math.NonlinLeastSquares

Properties

AbsoluteTolerance

public double AbsoluteTolerance {get; set; }
The absolute function tolerance. A double scalar value specifying the absolute function tolerance.

By default, 1.0e-32 is used as the absolute function tolerance.

Digits

virtual public int Digits {get; set; }
The number of good digits in the function. An int specifying the number of good digits in the user supplied function which defines the least-squares problem.

By default, the number of good digits is set to 7.

ErrorStatus

public int ErrorStatus {get; }
Get information about the performance of NonlinLeastSquares. An int specifying information about convergence.
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All convergence tests were met.</td>
</tr>
<tr>
<td>1</td>
<td>Scaled step tolerance was satisfied. The current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or <strong>StepTolerance</strong> is too big.</td>
</tr>
<tr>
<td>2</td>
<td>Scaled actual and predicted reductions in the function are less than or equal to the relative function convergence tolerance <strong>RelativeTolerance</strong>.</td>
</tr>
<tr>
<td>3</td>
<td>Iterates appear to be converging to a noncritical point. Incorrect gradient information, a discontinuous function, or stopping tolerances being too tight may be the cause.</td>
</tr>
<tr>
<td>4</td>
<td>Five consecutive steps with the maximum stepsize have been taken. Either the function is unbounded below, or has a finite asymptote in some direction, or the maximum stepsize is too small.</td>
</tr>
</tbody>
</table>

See Also:  
Imsl.Math.NonlinLeastSquares.RelativeTolerance (p. 131),  

**FalseConvergenceTolerance**

    public double FalseConvergenceTolerance {get; set; }

The false convergence tolerance. A **double** scalar value specifying the false convergence tolerance.

By default, 100.0e-16 is used as the false convergence tolerance.

**GradientTolerance**

    public double GradientTolerance {get; set; }

The gradient tolerance used to compute the gradient. A **double** specifying the gradient tolerance used to compute the gradient.

By default, the cube root of machine precision squared is used to compute the gradient.

**InitialTrustRegion**

    public double InitialTrustRegion {get; set; }

The initial trust region radius. A **double** scalar value specifying the initial trust region radius.

By default, **InitialTrustRegion** is set based on the initial scaled Cauchy step.

**MaximumIterations**

    public int MaximumIterations {get; set; }

The maximum number of iterations allowed. An **int** specifying the maximum number of iterations allowed.

By default, the maximum number of iterations is set to 100.

**MaximumStepsize**

    public double MaximumStepsize {get; set; }
The maximum allowable stepsize to use. A nonnegative double value specifying the maximum allowable stepsize.

By default, the maximum stepsize is set to a default value based on a scaled Guess.

RelativeTolerance

public double RelativeTolerance {get; set; }

The relative function tolerance. A double scalar value specifying the relative function tolerance.

By default, 1.0e-20 is used as the relative function tolerance.

StepTolerance

public double StepTolerance {get; set; }

The step tolerance used to step between two points. A double scalar value specifying the step tolerance used to step between two points.

By default, the cube root of machine precision is used as the step tolerance.

Constructor

NonlinLeastSquares

public NonlinLeastSquares(int m, int n)

Creates an object to solve a nonlinear least squares problem.

m  The number of functions
n  The number of variables. n must be less than or equal to m.

Methods

SetFscale

public void SetFscale(double[] fscale)

Sets the diagonal scaling matrix for the functions.

By default, the identity is used.

fscale  A double array specifying the diagonal scaling matrix for the functions.

System.ArgumentException is thrown if any of the elements of fscale is less than or equal to 0

SetGuess

public void SetGuess(double[] guess)

Sets the initial guess of the minimum point of the input function.

By default, an initial guess of 0.0 is used.

guess  A double array specifying the initial guess of the minimum point of the input function.
SetXscale

```csharp
public void SetXscale(double[] xscale)
```
Set the diagonal scaling matrix for the variables.

By default, the identity is used.

`xscale`  A double array specifying the diagonal scaling matrix for the variables.

**System.ArgumentException** is thrown if any of the elements of `xscale` is less than or equal to 0

Solve

```csharp
public double[] Solve(Imsl.Math.NonlinLeastSquares.IFunction f)
```
Solve a nonlinear least-squares problem using a modified Levenberg-Marquardt algorithm and a Jacobian.

`f`  User supplied NonlinLeastSquares.IFunction that defines the least-squares problem. If `f` implements IJacobian then its Jacobian is used. Otherwise, a finite difference Jacobian is used.

Returns — A double array of length n containing the approximate solution.

**Imsl.Math.TooManyIterationsException** is thrown if the number of iterations exceeds MaximumIterations, MaximumIterations is set to 100 by default

**Description**

*NonlinLeastSquares* is based on the MINPACK routine LMDIF by More et al. (1980). It uses a modified Levenberg-Marquardt method to solve nonlinear least squares problems. The problem is stated as follows:

\[
\min_{x \in \mathbb{R}^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2
\]

where \( m \geq n \), \( F : \mathbb{R}^n \rightarrow \mathbb{R}^m \), and \( f_i(x) \) is the i-th component function of \( F(x) \). From a current point, the algorithm uses the trust region approach:

\[
\min_{x_n \in \mathbb{R}^n} \| F(x_c) + J(x_c) (x_n - x_c) \|_2
\]

subject to

\[
\| x_n - x_c \|_2 \leq \delta_c
\]

to get a new point \( x_n \), which is computed as
\[ x_n = x_c - \left( J(x_c)^T J(x_c) + \mu_c I \right)^{-1} J(x_c)^T F(x_c) \]

where \( \mu_c = 0 \) if \( \delta_c \geq \| J(x_c)^T J(x_c) \|_2^{-1} \| J(x_c)^T F(x_c) \|_2 \) and \( \mu_c > 0 \) otherwise. \( F(x_c) \) and \( J(x_c) \) are the function values and the Jacobian evaluated at the current point \( x_c \). This procedure is repeated until the stopping criteria are satisfied. For more details, see Levenberg (1944), Marquardt (1963), or Dennis and Schnabel (1983, Chapter 10).

A finite-difference method is used to estimate the Jacobian when the user supplied function, \( f \), defines the least-squares problem. Whenever the exact Jacobian can be easily provided, \( f \) should implement \texttt{NonlinLeastSquares.Jacobian}.

**Example 1: Nonlinear least-squares problem**

A nonlinear least-squares problem is solved using a finite-difference Jacobian.

```csharp
using System;
using Imsl.Math;

public class NonlinLeastSquaresEx1 : NonlinLeastSquares.IFunction
{
    public void F(double[] x, double[] f)
    {
        f[0] = 10.0 * (x[1] - x[0] * x[0]);
        f[1] = 1.0 - x[0];
    }

    public static void Main(String[] args)
    {
        int m = 2;
        int n = 2;
        double[] x = new double[m];
        NonlinLeastSquares zs = new NonlinLeastSquares(m, n);
        zs.SetGuess(new double[]{ -1.2, 1.0 });
        zs.SetXscale(new double[]{ 1.0, 1.0 });
        zs.SetFscale(new double[]{ 1.0, 1.0 });
        x = zs.Solve(new NonlinLeastSquaresEx1());
        for (int k = 0; k < n; k++)
        {
            Console.Out.WriteLine("x[" + k + "] = " + x[k]);
        }
    }
}
```
Example 2: Nonlinear least-squares problem

A nonlinear least-squares problem is solved using a user-supplied Jacobian.

```csharp
using System;
using Imsl.Math;

class NonlinLeastSquaresEx2 : NonlinLeastSquares.IJacobian
{
    public void F(double[] x, double[] f)
    {
        f[0] = 10.0 * (x[1] - x[0] * x[0]);
        f[1] = 1.0 - x[0];
    }

    public void Jacobian(double[] x, double[,] fjac)
    {
        fjac[0, 0] = -20.0 * x[0];
        fjac[1, 0] = 10.0;
        fjac[0, 1] = -1.0;
        fjac[1, 1] = 0.0;
    }

    public static void Main(string[] args)
    {
        int m = 2;
        int n = 2;
        double[] x = new double[n];
        NonlinLeastSquares zs = new NonlinLeastSquares(m, n);
        zs.SetGuess(new double[]{-1.2, 1.0});
        zs.SetXscale(new double[]{1.0, 1.0});
        zs.SetFscale(new double[]{1.0, 1.0});
        x = zs.Solve(new NonlinLeastSquaresEx2());
        for (int k = 0; k < n; k++)
        {
            Console.WriteLine("x[" + k + "] = " + x[k]);
        }
    }
}
```

Output

\[
x[0] = 1 \\
x[1] = 1
\]
NonlinLeastSquares.IFunction Interface

Interface for the user supplied nonlinear least-squares function.

```java
public interface Imsl.Math.NonlinLeastSquares.IFunction
```

**Method**

```java
abstract public void F(double[] x, double[] fvalue)
```
User supplied nonlinear least-squares function.

- `x` A `double` array containing the point at which the function is to be evaluated. The contents of this array must not be altered by this function.
- `fvalue` A `double` array which, on return, contains the function value at `x`.

NonlinLeastSquares.IJacobian Interface

Interface for the user supplied nonlinear least squares function and its Jacobian.

```java
public interface Imsl.Math.NonlinLeastSquares.IJacobian implements
```

**Method**

```java
abstract public void Jacobian(double[] x, double[,] jvalue)
```
Jacobian of the user supplied nonlinear least squares function.

- `x` A `double` array containing the point at which the Jacobian of the function is to be evaluated.
- `jvalue` A `double` matrix which, on return, contains the value of the Jacobian, of the function, at `x`.

LinearProgramming Class

Solves a linear programming problem using the revised simplex algorithm.
public class Imsl.Math.LinearProgramming implements Properties

MaximumIterations
   public int MaximumIterations {get; set; }
   Sets the maximum number of iterations. Default is set to 10000. An int scalar specifying the maximum number of iterations.

ObjectiveValue
   public double ObjectiveValue {get; }
   Returns the optimal value of the objective function. A double scalar containing the optimal value of the objective function.

Constructor

LinearProgramming
   public LinearProgramming(double[,] a, double[] b, double[] c)
   Constructor variables of type double.
   a  A double matrix with coefficients of the constraints
   b  A double array containing the right-hand side of the constraints.
   c  A double array containing the coefficients of the objective function.
   System.ArgumentException is thrown if the dimensions of a, b.length, and c.length are not consistent

Methods

Clone
   Final public Object Clone()
   Creates and returns a copy of this object.
   Returns — A copy of this object.

GetDualSolution
   public double[] GetDualSolution()
   Returns the dual solution.
   Returns — A double array containing the dual solution of the linear programming problem.

GetSolution
   public double[] GetSolution()
   Returns the solution x of the linear programming problem.
   Returns — A double array containing the solution x of the linear programming problem.
SetConstraintType

public void SetConstraintType(int[] constraintType)
Sets the types of general constraints in the matrix a.
Let \( r_i = a_{i1}x_1 + \cdots + a_{in}x_n \)

<table>
<thead>
<tr>
<th>constraintType</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( r_i = b_i )</td>
</tr>
<tr>
<td>1</td>
<td>( r_i \leq b_u i )</td>
</tr>
<tr>
<td>2</td>
<td>( r_i \geq b_l )</td>
</tr>
<tr>
<td>3</td>
<td>( b_l \leq r_i \leq b_u i )</td>
</tr>
</tbody>
</table>

**constraintType**  A int array containing the types of general constraints.

SetLowerBound

public void SetLowerBound(double[] lowerBound)
Sets the lower bounds on the variables.
If there is no lower bound on a variable, then 10e30 should be set as the lower bound.

**lowerBound**  A double array containing the lower bounds on the variables.

SetUpperBound

public void SetUpperBound(double[] upperBound)
Sets the upper bound on the variables.
If there is no upper bound on a variable, then -10e30 should be set as the upper bound.

**upperBound**  A double array containing the upper bound on the variables.

SetUpperLimit

public void SetUpperLimit(double[] upperLimit)
Sets the upper limit of the constraints.
If no such constraint exists, then \( b_u \) is not needed.

**upperLimit**  A double array containing the upper limit of the constraints that have both the lower and the upper bounds.

Solve

public void Solve()
Solves the problem using the revised simplex algorithm.

Imsl.Math.BoundsInconsistentException is thrown if the bounds are inconsistent
Imsl.Math.ProblemInfeasibleException is thrown if there is no feasible solution to the problem
Imsl.Math.ProblemUnboundedException is thrown if there is no finite solution to the problem
Imsl.Math.NumericDifficultyException is thrown if there is a numerical problem during the solution
**Description**

Class **LinearProgramming** uses a revised simplex method to solve linear programming problems, i.e., problems of the form

\[
\min_{x \in \mathbb{R}^n} c^T x
\]

subject to

\[
b_l \leq A x \leq b_u
\]

\[
x_l \leq x \leq x_u
\]

where \(c\) is the objective coefficient vector, \(A\) is the coefficient matrix, and the vectors \(b_l, b_u, x_l,\) and \(x_u\) are the lower and upper bounds on the constraints and the variables, respectively.

For a complete description of the revised simplex method, see Murtagh (1981) or Murty (1983).

**Example 1: Linear Programming**

The linear programming problem in the standard form

\[
\min f(x) = -x_1 - 3x_2
\]

subject to:

\[
\begin{align*}
x_1 + x_2 + x_3 &= 1.5 \\
x_1 + x_2 - x_4 &= 0.5 \\
x_1 + x_5 &= 1.0 \\
x_2 + x_6 &= 1.0 \\
x_i &\geq 0, \text{ for } i = 1, \ldots, 6
\end{align*}
\]

is solved.

```csharp
using System;
using Imsl.Math;

class LinearProgrammingEx1
{
    public static void Main(string[] args)
    {
        double[,] a = {{1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0},
```
Example 2: Linear Programming

The linear programming problem

\[
\min f(x) = -x_1 - 3x_2
\]

subject to:

\[
\begin{align*}
0.5 & \leq x_1 + x_2 \leq 1.5 \\
0 & \leq x_1 \leq 1.0 \\
0 & \leq x_2 \leq 1.0
\end{align*}
\]

using System;
using Imsl.Math;

public class LinearProgrammingEx2
{
    public static void Main(String[] args)
    {
        int[] constraintType = new int[]{3};
        double[] upperBound = new double[]{1.0, 1.0};
        double[,] a = {{1.0, 1.0}};
        double[] b = new double[]{0.5};
        double[] upperLimit = new double[]{1.5};
        LinearProgramming zf = new LinearProgramming(a, b, c);
        zf.Solve();
        new PrintMatrix("Solution").Print(zf.GetSolution());
    }
}
```csharp
double[] c = new double[]{- 1.0, - 3.0};
LinearProgramming zf = new LinearProgramming(a, b, c);
zf.SetUpperLimit(upperLimit);
zf.SetConstraintType(constraintType);
zf.SetUpperBound(upperBound);
zf.Solve();
new PrintMatrix("Solution").Print(zf.GetSolution());
new PrintMatrix("Dual Solution").Print(zf.GetDualSolution());
Console.Out.WriteLine("Optimal Value = " + zf.ObjectiveValue);
```

Output

Solution

0 0 0.5
1 1

Dual Solution

0 0 -1

Optimal Value = -3.5

---

**QuadraticProgramming Class**

Solves the convex quadratic programming problem subject to equality or inequality constraints.

```csharp
public class Imsl.Math.QuadraticProgramming

Property

NoMoreProgress

public bool NoMoreProgress {get; }
Contains status of true or false if computer rounding error is inhibiting improvement in the objective function. Is true if due to computer rounding error, a change in the variables fails to improve the objective function. Usually the solution is close to optimum.
```
Constructor

QuadraticProgramming

public QuadraticProgramming(double[,] h, double[] g, double[,] aEquality,
 double[] bEquality, double[,] aInequality, double[] bInequality)
Solve a quadratic programming problem.

h  A square array containing the Hessian. It must be positive definite.
g  A double array containing the coefficients of the linear term of the objective function.
aEquality  A rectangular matrix containing the equality constraints. It can be null if there are no equality constraints.
bEquality  A double array containing the right-side of the equality constraints. It can be null if there are no equality constraints.
aInequality  A rectangular matrix containing the inequality constraints. It can be null if there are no inequality constraints.
bInequality  A double array containing the right-side of the inequality constraints. It can be null if there are no inequality constraints.

Imsl.Math.InconsistentSystemException is thrown if the problem is inconsistent.

Methods

GetDualSolution

public double[] GetDualSolution()
Returns the dual (Lagrange multipliers).

Returns — A double array containing the dual.

GetSolution

public double[] GetSolution()
Returns the solution.

Returns — A double array containing the unique solution.

Description

Class QuadraticProgramming is based on M.J.D. Powell’s implementation of the Goldfarb and Idnani dual quadratic programming (QP) algorithm for convex QP problems subject to general linear equality/inequality constraints (Goldfarb and Idnani 1983); i.e., problems of the form

$$\min_{x \in \mathbb{R}^n} g^T x + \frac{1}{2} x^T H x$$

subject to
given the vectors $b_1$, $b_2$, and $g$, and the matrices $H$, $A_1$, and $A_2$. $H$ is required to be positive definite. In this case, a unique $x$ solves the problem or the constraints are inconsistent. If $H$ is not positive definite, a positive definite perturbation of $H$ is used in place of $H$. For more details, see Powell (1983, 1985).

If a perturbation of $H$, $H + \alpha I$, is used in the $QP$ problem, then $H + \alpha I$ also should be used in the definition of the Lagrange multipliers.

**Example 1: Solve a Quadratic Programming Problem**

The quadratic programming problem is to minimize

$$
x_0^2 + x_1^2 + x_2^2 + x_3^2 + x_4^2 - 2x_1x_2 - 2x_3x_4 - 2x_0$$

subject to

$$
x_0 + x_1 + x_2 + x_3 + x_4 = 5$$

$$
x_2 - 2x_3 - 2x_4 = -3$$

using System;
using Imsl.Math;

public class QuadraticProgrammingEx1
{
    public static void Main(String[] args)
    {
        double[,] h = {
            {2, 0, 0, 0, 0},
            {0, 2, -2, 0, 0},
            {0, -2, 2, 0, 0},
            {0, 0, 0, 2, -2},
            {0, 0, 0, -2, 2}
        };
        double[,] aeq = {
            {1, 1, 1, 1, 1},
            {0, 0, 1, -2, -2}
        };
        double[] beq = new double[] {5, -3};
double[] g = new double[]{-2, 0, 0, 0, 0};
QuadraticProgramming qp = new QuadraticProgramming(h, g, aeq, beq, null, null);

// Print the solution and its dual
new PrintMatrix("x").Print(qp.GetSolution());
new PrintMatrix("dual").Print(qp.GetDualSolution());

Output

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>-1.18329135783152E-32</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Example 2: Solve a Quadratic Programming Problem

The quadratic programming problem is to minimize

\[ x_0^2 + x_1^2 + x_2^2 \]

subject to

\[ x_0 + 2x_1 - x_2 = 4 \]

\[ x_0 - x_1 + x_2 = -2 \]
public static void Main(String[] args)
{
    double[,] h = {
        {2, 0, 0},
        {0, 2, 0},
        {0, 0, 2}
    };
    double[,] aeq = {
        {1, 2, -1},
        {1, -1, 1}
    };
    double[] beq = new double[]{4, -2};
    double[] g = new double[]{0, 0, 0};

    QuadraticProgramming qp =
        new QuadraticProgramming(h, g, aeq, beq, null, null);

    // Print the solution and its dual
    new PrintMatrix("x").Print(qp.GetSolution());
    new PrintMatrix("dual").Print(qp.GetDualSolution());
}

Output

x
  0 0.285714285714286
  1 1.42857142857143
  2 -0.857142857142857

dual
  0 1.14285714285714
  1 -0.571428571428572
  2 0

MinConGenLin Class

Minimizes a general objective function subject to linear equality/inequality constraints.

public class Imsl.Math.MinConGenLin
Properties

FinalActiveConstraintsNum

```csharp
public int FinalActiveConstraintsNum {get; }
```

Returns the final number of active constraints. An int scalar containing the final number of active constraints.

ObjectiveValue

```csharp
public double ObjectiveValue {get; }
```

Returns the value of the objective function. A double scalar containing the value of the objective function.

Tolerance

```csharp
public double Tolerance {get; set; }
```

The nonnegative tolerance on the first order conditions at the calculated solution. A double scalar containing the tolerance.

Constructor

```csharp
public MinConGenLin(IFunction fcn, int nvar, int ncon, int neq, double[] a, double[] b, double[] lowerBound, double[] upperBound)
```

Constructor for MinConGenLin.

- `fcn` The user-supplied MinConGenLin.IFunction to be minimized.
- `nvar` An int scalar containing the number of variables.
- `ncon` An int scalar containing the number of linear constraints (excluding simple bounds).
- `neq` An int scalar containing the number of linear equality constraints.
- `a` A double array containing the equality constraint gradients in the first neq rows followed by the inequality constraint gradients. `a.length = ncon * nvar`.
- `b` A double array containing the right-hand sides of the linear constraints.
- `lowerBound` A double array containing the lower bounds on the variables. `lowerBound.length = nvar`.
- `upperBound` A double array containing the upper bounds on the variables. `upperBound.length = nvar`.

System.ArgumentException is thrown if the dimensions of `nvar`, `ncon`, `neq`, `a.length`, `b.length`, `lowerBound.length` and `upperBound.length` are not consistent.

Methods

GetFinalActiveConstraints
public int[] GetFinalActiveConstraints()
Returns the indices of the final active constraints.
Returns — An int array containing the indices of the final active constraints.

GetLagrangeMultiplierEstimate
public double[] GetLagrangeMultiplierEstimate()
Returns the Lagrange multiplier estimates of the final active constraints.
Returns — A double array containing the Lagrange multiplier estimates of the final active constraints.

GetSolution
public double[] GetSolution()
Returns the computed solution.
Returns — A double array containing the computed solution.

SetGuess
public void SetGuess(double[] guess)
Sets an initial guess of the solution.
guess — A double array containing an initial guess.

Solve
public void Solve()
Minimizes a general objective function subject to linear equality/inequality constraints.

Imsl.Math.ConstraintsInconsistentException is thrown if the constraints are inconsistent.
Imsl.Math.VarBoundsInconsistentException is thrown if the bounds on the variables are inconsistent.
Imsl.Math.ConstraintsNotSatisfiedException is thrown if a solution satisfying the constraints could not be found.
Imsl.Math.EqualityConstraintsException is thrown if the variables are determined by the constraints.

Description
The class MinConGenLin is based on M.J.D. Powell’s TOLMIN, which solves linearly constrained optimization problems, i.e., problems of the form

\[ \min f(x) \]

subject to

\[ A_1 x = b_1 \]
\[ A_2 x \leq b_2 \]
\[ x_l \leq x \leq x_u \]

given the vectors \( b_1, b_2, x_l, \) and \( x_u \) and the matrices \( A_1 \) and \( A_2 \).

The algorithm starts by checking the equality constraints for inconsistency and redundancy. If the equality constraints are consistent, the method will revise \( x^0 \), the initial guess, to satisfy

\[ A_1 x = b_1 \]

Next, \( x^0 \) is adjusted to satisfy the simple bounds and inequality constraints. This is done by solving a sequence of quadratic programming subproblems to minimize the sum of the constraint or bound violations.

Now, for each iteration with a feasible \( x^k \), let \( J_k \) be the set of indices of inequality constraints that have small residuals. Here, the simple bounds are treated as inequality constraints. Let \( I_k \) be the set of indices of active constraints. The following quadratic programming problem

\[
\min f(x^k) + d^T \nabla f(x^k) + \frac{1}{2} d^T B^k d
\]

subject to

\[
a_j d = 0, \ j \in I_k
\]

\[
a_j d \leq 0, \ j \in J_k
\]

is solved to get \((d^k, \lambda^k)\) where \( a_j \) is a row vector representing either a constraint in \( A_1 \) or \( A_2 \) or a bound constraint on \( x \). In the latter case, the \( a_j = e_j \) for the bound constraint \( x_i \leq (x_u)_i \) and \( a_j = -e_i \) for the constraint \(-x_i \leq (x_l)_i\). Here, \( e_i \) is a vector with 1 as the \( i \)-th component, and zeros elsewhere. Variables \( \lambda^k \) are the Lagrange multipliers, and \( B^k \) is a positive definite approximation to the second derivative \( \nabla^2 f(x^k) \).

After the search direction \( d^k \) is obtained, a line search is performed to locate a better point. The new point \( x^{k+1} = x^k + \alpha^k d^k \) has to satisfy the conditions

\[
f(x^k + \alpha^k d^k) \leq f(x^k) + 0.1 \alpha^k (d^k)^T \nabla f(x^k)
\]
and

\[(d^k)^T \nabla f(x^k + \alpha^k d^k) \geq 0.7(d^k)^T \nabla f(x^k)\]

The main idea in forming the set \(J_k\) is that, if any of the equality constraints restricts the step-length \(\alpha^k\), then its index is not in \(J_k\). Therefore, small steps are likely to be avoided. Finally, the second derivative approximation \(B^K\), is updated by the BFGS formula, if the condition

\[(d^K)^T \nabla f(x^k + \alpha^k d^k) - \nabla f(x^k) > 0\]

holds. Let \(x^k \leftarrow x^{k+1}\), and start another iteration. The iteration repeats until the stopping criterion

\[\|\nabla f(x^k) - A^k \lambda^K\|_2 \leq \tau\]

is satisfied. Here \(\tau\) is the supplied tolerance. For more details, see Powell (1988, 1989).

**Example 1: Linear Constrained Optimization**

The problem

\[
\min f(x) = x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 2x_2x_3 - 2x_4x_5 - 2x_1
\]

subject to

\[
x_1 + x_2 + x_3 + x_4 + x_5 = 5
\]

\[
x_3 - 2x_4 - 2x_5 = -3
\]

\[0 \leq x \leq 10\]

is solved.
using System;
using Imsl.Math;

public class MinConGenLinEx1 : MinConGenLin.IFunction
{
    public double F(double[] x)
    {
            x[4] - 2.0 * x[0];
    }

    public static void Main(String[] args)
    {
        int neq = 2;
        int ncon = 2;
        int nvar = 5;
        double[] a = new double[]{1.0, 1.0, 1.0, 1.0, 1.0,
                                   0.0, 0.0, 1.0, -2.0, -2.0};
        double[] b = new double[]{5.0, -3.0};
        double[] xlb = new double[]{0.0, 0.0, 0.0, 0.0, 0.0};
        double[] xub = new double[]{10.0, 10.0, 10.0, 10.0, 10.0};

        MinConGenLin.IFunction fcn = new MinConGenLinEx1();
        MinConGenLin zf = new MinConGenLin(fcn, nvar, ncon, neq, a, b,
                                             xlb, xub);
        zf.Solve();
        new PrintMatrix("Solution").Print(zf.GetSolution());
    }
}

Output

Solution
0
0 1
1 1
2 1
3 1
4 1

Example 2: Linear Constrained Optimization

The problem

\[ \min f(x) = -x_0x_1x_2 \]

subject to
\[-x_0 - 2x_1 - 2x_2 \leq 0\]
\[x_0 + 2x_1 + 2x_2 \leq 72\]
\[0 \leq x_0 \leq 20\]
\[0 \leq x_1 \leq 11\]
\[0 \leq x_2 \leq 42\]

is solved with an initial guess of \(x_0 = 10\), \(x_1 = 10\) and \(x_2 = 10\).

```csharp
using System;
using Imsl.Math;

class MinConGenLinEx2 : MinConGenLin.IGradient
{
    public double F(double[] x)
    {
        return -x[0] * x[1] * x[2];
    }

    public void Gradient(double[] x, double[] g)
    {
        g[0] = -x[1] * x[2];
        g[1] = -x[0] * x[2];
        g[2] = -x[0] * x[1];
    }

    public static void Main(string[] args)
    {
        int neq = 0;
        int ncon = 2;
        int nvar = 3;
        double[] a = new double[]{1.0, -2.0, -2.0, 1.0, 2.0, 2.0};
        double[] xlb = new double[]{0.0, 0.0, 0.0};
        double[] xub = new double[]{20.0, 11.0, 42.0};
        double[] b = new double[]{0.0, 72.0};

        MinConGenLin.IGradient fcn = new MinConGenLinEx2();
        MinConGenLin zf = new MinConGenLin(fcn, nvar, ncon, neq, a, b, xlb, xub);
        zf.SetGuess(new double[]{10.0, 10.0, 10.0});
        zf.Solve();
        new PrintMatrix("Solution").Print(zf.GetSolution());
    }
}
```
Console.Out.WriteLine("Objective value = " + 
    zf.ObjectiveValue);
}
}

Output

Solution
0
0 20
1 11
2 15

Objective value = -3300

MinConGenLin.IFunction Interface

Public interface for the user-supplied function to evaluate the function to be minimized.

public interface Imsl.Math.MinConGenLin.IFunction

Method

F

abstract public double F(double[] x)
Public interface for the function to be minimized.

x A double array, the point at which the function is evaluated. x.length equals the number of variables.

Returns — A double scalar, the function value at x.

MinConGenLin.IGradient Interface

Public interface for the user-supplied function to compute the gradient.

public interface Imsl.Math.MinConGenLin.IGradient implements
Method

Gradient

abstract public void Gradient(double[] x, double[] g)
Public interface for the user-supplied function to compute the gradient at point x.

\( x \) A double array, the point at which the gradient is evaluated. \( x.length \) equals the number of variables.

\( g \) A double array which, on return, contains the values of the gradient of the objective function.

BoundedLeastSquares Class

Solves a nonlinear least-squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm.

public class Imsl.Math.BoundedLeastSquares

Properties

AbsoluteTolerance

public double AbsoluteTolerance {get; set; }
The absolute function tolerance. A double scalar containing the absolute function tolerance.

Digits

public int Digits {get; set; }
The number of good digits in the function. A int scalar containing the number of good digits.

GradientTolerance

public double GradientTolerance {get; set; }
The scaled gradient tolerance. A double scalar containing the scaled gradient tolerance.

MaximumFunctionEvals

public int MaximumFunctionEvals {get; set; }
The maximum number of function evaluations. A int scalar containing the maximum number of function evaluations.

MaximumIterations

public int MaximumIterations {get; set; }
The maximum number of iterations. A int scalar containing the maximum number of iterations.
MaximumJacobianEvals

```csharp
public int MaximumJacobianEvals {get; set; }
```

The maximum number of Jacobian evaluations. A `int` scalar containing the maximum number of Jacobian evaluations.

MaximumStepsize

```csharp
public double MaximumStepsize {get; set; }
```

The maximum allowable step size. A `double` scalar containing the maximum allowable step size.

RelativeTolerance

```csharp
public double RelativeTolerance {get; set; }
```

The relative function tolerance. A `double` scalar containing the relative function tolerance.

ScaledStepTolerance

```csharp
public double ScaledStepTolerance {get; set; }
```

The scaled step tolerance. A `double` scalar containing the scaled step tolerance.

TrustRegion

```csharp
public double TrustRegion {get; set; }
```

The size of initial trust region radius. A `double` scalar containing the initial trust region radius.

**Constructor**

BoundedLeastSquares

```csharp
public BoundedLeastSquares(Imsl.Math.BoundedLeastSquares.IFunction f, int mFunctions, int nVariables, int boundType, double[] lowerBound, double[] upperBound)
```

Constructor for `BoundedLeastSquares`.

- `f` The user-supplied `BoundedLeastSquares.IFunction` to be minimized.
- `mFunctions` A `int` scalar containing the number of functions.
- `nVariables` A `int` scalar containing the number of variables.
- `boundType` A `int` scalar containing the types of bounds on the variable.

<table>
<thead>
<tr>
<th>boundType</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>User will supply all the bounds.</td>
</tr>
<tr>
<td>1</td>
<td>All variables are nonnegative.</td>
</tr>
<tr>
<td>2</td>
<td>All variables are nonpositive.</td>
</tr>
<tr>
<td>3</td>
<td>User supplies only the bounds on first variable, all other variables will have the same bounds.</td>
</tr>
</tbody>
</table>

- `lowerBound` A `double` array containing the lower bounds on the variables.
- `upperBound` A `double` array containing the upper bounds on the variables.

`System.ArgumentException` is thrown if the dimensions of `mFunctions`, `nVariables`, `boundType`, `lowerBound.length` and `upperBound.length` are not consistent.
Methods

GetJacobianSolution

public double[,] GetJacobianSolution()
Returns the Jacobian at the approximate solution.
Returns — A mFunctions x nVariables double matrix containing the Jacobian at the approximate solution.

GetResiduals

public double[] GetResiduals()
Returns the residuals at the approximate solution.
Returns — A double array containing the residuals at the approximate solution.

GetSolution

public double[] GetSolution()
Returns the solution.
Returns — A double array containing the computed solution.

SetFscale

public void SetFscale(double[] fscale)
Sets the diagonal scaling matrix for the functions.
The i-th component of fscale is a positive scalar specifying the reciprocal magnitude of the i-th component function of the problem. Default: fscale[] = 1

fscale  A double array containing the diagonal scaling for the functions.

SetGuess

public void SetGuess(double[] guess)
Sets the initial guess of the solution.
guess  A double array containing an initial guess.

SetInternalScale

public void SetInternalScale()
The internal variable scaling option.
With this option, the values for xscale are set internally.

SetXscale

public void SetXscale(double[] xscale)
The scaling vector for the variables.
Argument xscale is used mainly in scaling the gradient and the distance between two points. See GradientTolernce and ScaledStepTolerance for more details. Default: xscale[] = 1

xscale  A double array containing the scaling vector for the variables.
public void solve()
Solves a nonlinear least-squares problem subject to bounds on the variables using a
modified Levenberg-Marquardt algorithm.

Imsl.Math.FalseConvergenceException is thrown if there is a problem with
convergence.

Description

Class BoundedLeastSquares uses a modified Levenberg-Marquardt method and an active set
strategy to solve nonlinear least-squares problems subject to simple bounds on the variables.
The problem is stated as follows:

$$
\min \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2
$$

subject to

$$
l \leq x \leq u
$$

here $m \geq n$, $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$, and $f_i(x)$ is the $i$-th component function of $F(x)$. From a given
starting point, an active set $IA$, which contains the indices of the variables at their bounds, is
built. A variable is called a "free variable" if it is not in the active set. The routine then
computes the search direction for the free variables according to the formula

$$
d = - (J^T J + \mu I)^{-1} J^T F
$$

where $\mu$ is the Levenberg-Marquardt parameter, $F = F(x)$, and $J$ is the Jacobian with respect
to the free variables. The search direction for the variables in $IA$ is set to zero. The trust region
approach discussed by Dennis and Schnabel (1983) is used to find the new point. Finally, the
optimality conditions are checked. The conditions are:

$$
\|g(x_i)\| \leq \varepsilon, l_i < x_i < u_i
$$

$$
g(x_i) < 0, x_i = u_i
$$

$$
g(x_i) > 0, x_i = l_i
$$

where $\varepsilon$ is a gradient tolerance. This process is repeated until the optimality criterion is
achieved.

The active set is changed only when a free variable hits its bounds during an iteration or the
optimality condition is met for the free variables but not for all variables in $IA$, the active set.
In the latter case, a variable that violates the optimality condition will be dropped out of $IA$. For
more details on the Levenberg-Marquardt method, see Levenberg (1944) or Marquardt
(1963). For more detail on the active set strategy, see Gill and Murray (1976).
Example 1: Bounded Least Squares

The nonlinear least-squares problem

\[
\min \frac{1}{2} \sum_{i=0}^{1} f_i(x)^2
\]

\[-2 \leq x_0 \leq 0.5\]

\[-1 \leq x_1 \leq 2\]

where

\[f_0(x) = 10(x_1 - x_0^2) \text{ and } f_1(x) = (1 - x_0)\]

is solved.

```csharp
using System;
using Imsl.Math;

public class BoundedLeastSquaresEx1 : BoundedLeastSquares.IFunction
{
    public void F(double[] x, double[] f)
    {
        f[0] = 10.0 * (x[1] - x[0] * x[0]);
        f[1] = 1.0 - x[0];
    }

    public static void Main(String[] args)
    {
        int m = 2;
        int n = 2;
        int ibtype = 0;
        double[] xlb = new double[]{-2.0, -1.0};
        double[] xub = new double[]{0.5, 2.0};
        BoundedLeastSquares.IFunction rosbck =
            new BoundedLeastSquaresEx1();
        BoundedLeastSquares zf =
            new BoundedLeastSquares(rosbck, m, n, ibtype, xlb, xub);
        zf.solve();
        new PrintMatrix("Solution").Print(zf.GetSolution());
    }
}
```

Output

Solution
Example 2: Bounded Least Squares

The nonlinear least-squares problem

\[
\min \frac{1}{2} \sum_{i=0}^{1} f_i(x)^2
\]

\[-2 \leq x_0 \leq 0.5\]

\[-1 \leq x_1 \leq 2\]

where

\[f_0(x) = 10(x_1 - x_0^2)\] and \[f_1(x) = (1 - x_0)\]

is solved. An initial guess \((-1.2, 1.0)\) is supplied, as well as the analytic Jacobian. The residual at the approximate solution is returned.

using System;
using Imsl.Math;

public class BoundedLeastSquaresEx2 : BoundedLeastSquares.IJacobian
{
    public void F(double[] x, double[] f)
    {
        f[0] = 10.0 * (x[1] - x[0] * x[0]);
        f[1] = 1.0 - x[0];
    }

    public void Jacobian(double[] x, double[] fjac)
    {
        fjac[0] = -20.0 * x[0];
        fjac[1] = 10.0;
        fjac[2] = -1.0;
        fjac[3] = 0.0;
    }

    public static void Main(String[] args)
    {
        int m = 2;
        int n = 2;
        int ibtype = 0;
    }
}
double[] xlb = new double[]{- 2.0, - 1.0};
double[] xub = new double[]{0.5, 2.0};

BoundedLeastSquares.IJacobian rosbck =
    new BoundedLeastSquaresEx2();
BoundedLeastSquares zf =
    new BoundedLeastSquares(rosbck, m, n, ibtype, xlb, xub);
zf.SetGuess(new double[]{- 1.2, 1.0});
zf.solve();
new PrintMatrix("Solution").Print(zf.GetSolution());
new PrintMatrix("Residuals").Print(zf.GetResiduals());

Output

Solution
  0
  0 0.5
  1 0.25

Residuals
  0
  0 0
  1 0.5

BoundedLeastSquares.IFunction Interface

Public interface for the user-supplied function to evaluate the function that defines the least-squares problem.

public interface Imsl.Math.BoundedLeastSquares.IFunction

Method

F
abstract public void F(double[] x, double[] fvalue)
Public interface for the user-supplied function to evaluate the function that defines the least-squares problem.

x  A double array, the point at which the function is to evaluated. x.length = nVariables.
fvalue  A double array, the function values at point x. f.Length = mFunctions.
BoundedLeastSquares.IJacobian Interface

Public interface for the user-supplied function to compute the Jacobian.

```java
public interface Imsl.Math.BoundedLeastSquares.IJacobian implements Method
{
  abstract public void Jacobian(double[] x, double[] fjac)
  // Public interface for the user-supplied function to compute the Jacobian.
  // x A double array, the point at which the Jacobian is to evaluate. x.length = nVariables.
  // fjac A double array which, on return, contains the computed Jacobian at the point x.
  // fjac.length = mFunctions x nVariables.
}
```

MinConNLP Class

General nonlinear programming solver.

```java
public class Imsl.Math.MinConNLP
{
  // Properties
  BindingThreshold
  public double BindingThreshold {get; set; }
  // The binding threshold for constraints. A double scalar value specifying the binding threshold for constraints.

  // In the initial phase of minimization a constraint is considered binding if
  // \[ \frac{g_i(x)}{\max(1, \|\nabla g_i(x)\|)} \leq BindingThreshold \quad i = M_c + 1, \ldots, M \]
  // Good values are between .01 and 1.0. If BindingThreshold is chosen too small then identification of the correct set of binding constraints may be delayed. Contrary, if BindingThreshold is too large, then the method will often escape to the full regularized SQP method, using individual slack variables for any active constraint, which is quite costly. For well scaled problems BindingThreshold = 1.0 is reasonable. By default, BindingThreshold is set to .5 * PenaltyBound.
}
```
BoundViolationBound

```csharp
public double BoundViolationBound {get; set; }
The amount by which bounds may be violated during numerical differentiation. A double scalar value specifying the amount by which bounds may be violated during numerical differentiation.

By default, BoundViolationBound is set to 1.0.
```

DifferentiationType

```csharp
public int DifferentiationType {get; set; }
The type of numerical differentiation to be used. An int scalar value specifying the type of numerical differentiation to be used. By default, DifferentiationType is set to 1.
```

<table>
<thead>
<tr>
<th>Value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Use a forward difference quotient with discretization stepsize (0.1 \left(\text{FunctionPrecision}^{1/2}\right)) componentwise relative. This is the default value used.</td>
</tr>
<tr>
<td>2</td>
<td>Use the symmetric difference quotient with discretization stepsize (0.1 \left(\text{FunctionPrecision}^{1/3}\right)) componentwise relative.</td>
</tr>
<tr>
<td>3</td>
<td>Use the sixth order approximation computing a Richardson extrapolation of three symmetric difference quotient values. This uses a discretization stepsize (0.01 \left(\text{FunctionPrecision}^{1/7}\right)).</td>
</tr>
</tbody>
</table>

FunctionPrecision

```csharp
public double FunctionPrecision {get; set; }
The relative precision of the function evaluation routine. A double scalar value specifying the relative precision of the function evaluation routine.

By default, FunctionPrecision is set to 2.2e-16.
```

GradientPrecision

```csharp
public double GradientPrecision {get; set; }
The relative precision in gradients. A double scalar value specifying the relative precision in gradients.

By default, GradientPrecision is set to 2.2e-16.
```

MaximumIterations

```csharp
public int MaximumIterations {get; set; }
The maximum number of iterations allowed. An int specifying the maximum number of iterations allowed.

By default, MaximumIterations is set to 200.
```

MultiplierError

```csharp
public double MultiplierError {get; set; }
The error allowed in the multipliers. A double scalar value specifying the error allowed in the multipliers.
A negative multiplier of an inequality constraint is accepted (as zero) if its absolute value is less than `MultiplierError`. By default, `MultiplierError` is set to $e^{2\log \epsilon/3}$.

**PenaltyBound**

```java
public double PenaltyBound {get; set; }
```

The universal bound for describing how much the unscaled penalty-term may deviate from zero. A `double` scalar value specifying the universal bound for describing how much the unscaled penalty-term may deviate from zero.

A small `PenaltyBound` diminishes the efficiency of the solver because the iterates then will follow the boundary of the feasible set closely. Conversely, a large `PenaltyBound` may degrade the reliability of the code. By default, `PenaltyBound` is set to 1.0.

**ScalingBound**

```java
public double ScalingBound {get; set; }
```

The scaling bound for the internal automatic scaling of the objective function. A `double` scalar value specifying the scaling variable for the problem function.

By default, `ScalingBound` is set to 1.0e4.

**ViolationBound**

```java
public double ViolationBound {get; set; }
```

Defines allowable constraint violations of the final accepted result. A `double` scalar value specifying the allowable constraint violations of the final accepted result.

Constraints are satisfied if $|g_i(x)| \leq \text{ViolationBound}$, and $g_i(x) \geq -\text{ViolationBound}$ respectively. By default, `ViolationBound` is set to $\min(\text{BindingThreshold}/10, \max(\text{epsdif}, \min(\text{BindingThreshold}/10, \max((1.e - 6)\text{BindingThreshold}, \text{small}_w))))$.

### Constructor

**MinConNLP**

```java
public MinConNLP(int mTotalConstraints, int mEqualityConstraints, int nVariables)
```

Nonlinear programming solver constructor.

- **mTotalConstraints** An `int` scalar value which defines the total number of constraints.
- **mEqualityConstraints** An `int` scalar value which defines the number of equality constraints.
- **nVariables** An `int` scalar value which defines the number of variables.

### Methods

**GetConstraintResiduals**

```java
public double[] GetConstraintResiduals()
```

Returns the constraint residuals.

Returns — A `double` array containing the constraint residuals.
GetLagrangeMultiplierEst
   public double[] GetLagrangeMultiplierEst()
   Returns the Lagrange multiplier estimates of the constraints.

   Returns — A double array containing the Lagrange multiplier estimates of the
   constraints.

SetGuess
   public void SetGuess(double[] guess)
   Sets the initial guess of the minimum point of the input function.

   By default, the elements of this array are set to x, (with the smallest value of ||x||_2) that
   satisfies the bounds.

   guess  A double array specifying the initial guess of the minimum point of the input
   function.

SetXlowerBound
   public void SetXlowerBound(double[] lower)
   Sets the lower bounds on the variables.

   By default, the elements of this array are set to -1.79e308.

   lower  A double array specifying the lower bounds on the variables.

SetXscale
   public void SetXscale(double[] scale)
   The internal scaling of the variables.

   The initial value given and the objective function and gradient evaluations, however, are
   always given in the original unscaled variables. The first internal variable is obtained by
   dividing the values x[i] by Xscale[i]. By default, Xscale[i] is set to 1.0.

   scale  A double array specifying the internal scaling of the variables.

   System.ArgumentException is thrown if Xscale[i] is less than or equal to 0.0

SetXupperBound
   public void SetXupperBound(double[] upper)
   Sets the upper bounds on the variables.

   By default, the elements of this array are set to 1.79e308.

   upper  A double array specifying the upper bounds on the variables.

Solve
   public double[] Solve(Imsl.Math.MinConNLP.IFunction f)
   Solve a general nonlinear programming problem using the successive quadratic
   programming algorithm with a finite-difference gradient or with a user-supplied gradient.
f  Defines the user-supplied MinConNLP.IFunction to be evaluated at a given point. f can be used to supply a MinConNLP.IGradient of the function. If f implements IGradient the user-supplied gradient is used. Otherwise, an attempt to solve the problem is made using a finite-difference gradient.

Returns — A double array containing the solution of the nonlinear programming problem.

Imsl.Math.ConstraintEvaluationException is thrown if a constraint evaluation returns an error.

Imsl.Math.ObjectiveEvaluationException is thrown if objective evaluation returns an error.

Imsl.Math.WorkingSetSingularException is thrown if

Imsl.Math.QPInfeasibleException is thrown if the working set is singular in dual extended QP.

Imsl.Math.PenaltyFunctionPointInfeasibleException is thrown if the penalty function point infeasible.

Imsl.Math.LimitingAccuracyException is thrown if limiting accuracy reached for a singular problem.

Imsl.Math.TooManyIterationsException is thrown if maximum number of iterations exceeded.

Imsl.Math.NoAcceptableStepsizeException is thrown if there is no acceptable stepsize.

Imsl.Math.BadInitialGuessException is thrown if the penalty function point infeasible for original problem.

Imsl.Math.IllConditionedException is thrown if the problem is singular or ill-conditioned.

Imsl.Math.SingularException is thrown if the problem is singular.

Imsl.Math.LinearlyDependentGradientsException is thrown if the working set gradients are linearly dependent.

Imsl.Math.TerminationCriteriaNotSatisfiedException is thrown if termination criteria are not satisfied.

Description

MinConNLP is based on the FORTRAN subroutine, DONLP2, by Peter Spellucci and licensed from TU Darmstadt. MinConNLP uses a sequential equality constrained quadratic programming method with an active set technique, and an alternative usage of a fully regularized mixed constrained subproblem in case of nonregular constraints (i.e. linear dependent gradients in the "working sets"). It uses a slightly modified version of the Pantoja-Mayne update for the Hessian of the Lagrangian, variable dual scaling and an improved Armijo-type stepsize algorithm. Bounds on the variables are treated in a gradient-projection like fashion. Details may be found in the following two papers:


The problem is stated as follows:

\[
\min_{x \in \mathbb{R}^n} f(x)
\]

subject to

\[
g_j(x) = 0, \text{ for } j = 1, \ldots, m_e
\]

\[
g_j(x) \geq 0, \text{ for } j = m_e + 1, \ldots, m
\]

\[
x_l \leq x \leq x_u
\]

where all problem functions are assumed to be continuously differentiable. Although default values are provided for optional input arguments, it may be necessary to adjust these values for some problems. Through the use of member functions, MinConNLP allows for several parameters of the algorithm to be adjusted to account for specific characteristics of problems. The DONLP2 Users Guide provides detailed descriptions of these parameters as well as strategies for maximizing the performance of the algorithm. In addition, the following are a number of guidelines to consider when using MinConNLP:

- A good initial starting point is very problem specific and should be provided by the calling program whenever possible. See method SetGuess.

- Gradient approximation methods can have an effect on the success of MinConNLP. Selecting a higher order approximation method may be necessary for some problems. See property DifferentiationType.

- If a two sided constraint \( l_i \leq g_i(x) \leq u_i \) is transformed into two constraints, \( g_{2i}(x) \geq 0 \) and \( g_{2i+1}(x) \geq 0 \), then choose BindingThreshold < \( 1/2 (u_i - l_i) / \max \{1, \|\nabla g_i(x)\|\} \), or at least try to provide an estimate for that value. This will increase the efficiency of the algorithm. See property BindingThreshold.

- The parameter ierr provided in the interface to the user supplied function F can be very useful in cases when evaluation is requested at a point that is not possible or reasonable. For example, if evaluation at the requested point would result in a floating point exception, then setting ierr to true and returning without performing the evaluation.
will avoid the exception. MinConNLP will then reduce the stepsize and try the step again. Note, if ierr is set to true for the initial guess, then an error is issued.

Example 1: Solving a general nonlinear programming problem

A general nonlinear programming problem is solved using a finite difference gradient.

```csharp
using System;
using Imsl.Math;

public class MinConNLPEx1 : MinConNLP.IFunction
{
    public double F(double[] x, int iact, bool[] ierr)
    {
        double result;
        ierr[0] = false;
        if (iact == 0)
        {
            result = (x[0] - 2.0) * (x[0] - 2e0) + 
            (x[1] - 1.0) * (x[1] - 1.0);
            return result;
        }
        else
        {
            switch (iact)
            {
                case 1:
                    result = (x[0] - 2.0 * x[1] + 1.0);
                    return result;
                case 2:
                    result = (-(x[0] * x[0]) / 4.0 - (x[1] * x[1]) + 1.0);
                    return result;
                default:
                    ierr[0] = true;
                    return 0.0;
            }
        }
    }

    public static void Main(String[] args)
    {
        int m = 2;
        int me = 1;
        int n = 2;
        MinConNLP minconnon = new MinConNLP(m, me, n);
        minconnon.SetGuess(new double[]{2.0, 2.0});
        double[] x = minconnon.Solve(new MinConNLPEx1());
    }
}
```
new PrintMatrix("x").Print(x);
}

Output

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.822875655532512</td>
</tr>
<tr>
<td>1</td>
<td>0.911437827766256</td>
</tr>
</tbody>
</table>

Example 2: Solving a general nonlinear programming problem

A general nonlinear programming problem is solved using a user-supplied gradient.

```csharp
using System;
using Imsl.Math;

class MinConNLPEx2 : MinConNLP.IGradient
{
    public double F(double[] x, int iact, bool[] ierr)
    {
        double result;
        ierr[0] = false;
        if (iact == 0)
        {
            result = (x[0] - 2.0) * (x[0] - 2.0) + (x[1] - 1.0) * (x[1] - 1.0);
            return result;
        }
        else
        {
            switch (iact)
            {
            case 1:
                result = (x[0] - 2.0 * x[1] + 1.0);
                return result;
            case 2:
                result = -(x[0] * x[0]) / 4.0 - (x[1] * x[1]) + 1.0;
                return result;
            default:
                ierr[0] = true;
                return 0.0;
            }
        }
    }
}
```
public void Gradient(double[] x, int iact, double[] result)
{
    if (iact == 0)
    {
        result[0] = 2.0 * (x[0] - 2.0);
        result[1] = 2.0 * (x[1] - 1.0);
        return;
    }
    else
    {
        switch (iact)
        {
            case 1:
                result[0] = 1.0;
                result[1] = -2.0;
                return;
            case 2:
                result[0] = -0.5 * x[0];
                result[1] = -2.0 * x[1];
                return;
        }
    }
}

public static void Main(String[] args)
{
    int m = 2;
    int me = 1;
    int n = 2;
    MinConNLP minconnon = new MinConNLP(m, me, n);
    minconnon.SetGuess(new double[]{2.0, 2.0});
    double[] x = minconnon.Solve(new MinConNLPEx2());
    new PrintMatrix("x").Print(x);
}

Output

x
0
0 0.82287565532512
1 0.911437827766256
MinConNLP.IFunction Interface

Public interface for the user supplied function to the MinConNLP object.

public interface Imsl.Math.MinConNLP.IFunction

Method

F

abstract public double F(double[] x, int iact, bool[] ierr)
Compute the value of the function at the given point.

x   An input double array, the point at which the objective function or constraint is to be evaluated.

iact   An input int value indicating whether evaluation of the objective function is requested or evaluation of a constraint is requested. If iact is zero, then an objective function evaluation is requested. If iact is nonzero then the value of iact indicates the index of the constraint to evaluate. (1 indicates the first constraint, 2 indicates the second, etc.)

ierr   An input/output boolean array of length 1. On input ierr[0] is set to false. If an error or other undesirable condition occurs during evaluation, then ierr[0] should be set to true. Setting ierr[0] to true will result in the step size being reduced and the step being tried again. (If ierr[0] is set to true for xguess, then an error is issued.)

Returns — A double. If iact is zero, then the value of the objective function at x is returned. If iact is nonzero, then the computed constraint value at the point x is returned.

MinConNLP.IGradient Interface

Public interface for the user supplied function to compute the gradient for MinConNLP object.

public interface Imsl.Math.MinConNLP.IGradient implements

Method

Gradient

abstract public void Gradient(double[] x, int iact, double[] result)
Computes the value of the gradient of the function at the given point.
x  An input double array, the point at which the gradient of the objective function or gradient of a constraint is to be evaluated.

iact  An input int value indicating whether evaluation of the objective function gradient is requested or evaluation of a constraint gradient is requested. If iact is zero, then an objective function gradient evaluation is requested. If iact is nonzero then the value of iact indicates the index of the constraint gradient to evaluate. (1 indicates the first constraint, 2 indicates the second, etc.)

result  A double array. If iact is zero, then the value of the objective function gradient at x is returned in result. If iact is nonzero, then the computed gradient of the requested constraint value at the point x is returned in result.
Chapter 9: Special Functions

Types

class Sfun ................................................................. 171
class Bessel ............................................................... 187

Sfun Class

Collection of special functions.

public class Imsl.Math.Sfun

Fields

EpsilonLarge
    public double EpsilonLarge
    The largest relative spacing for doubles.

EpsilonSmall
    public double EpsilonSmall
    The smallest relative spacing for doubles.

Methods

Asinh
    static public double Asinh(double x)
    Returns the hyperbolic arc sine of a double.

    x  A double value for which the hyperbolic arc sine is desired.
Returns — A **double** specifying the hyperbolic arc sine value.

Beta

```csharp
static public double Beta(double a, double b)
```

Returns the value of the Beta function.

The Beta function is defined to be

\[
\beta(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a + b)} = \int_0^1 t^{a-1}(1-t)^{b-1}dt
\]

See **Gamma** for the definition of \(\Gamma(x)\).

The method **Beta** requires that both arguments be positive.

- **a** A **double** value.
- **b** A **double** value.

Returns — A **double** value specifying the Beta function.

BetaIncomplete

```csharp
static public double BetaIncomplete(double x, double p, double q)
```

Returns the incomplete Beta function ratio.

The incomplete beta function is defined to be

\[
I_x(p, q) = \frac{\beta_x(p, q)}{\beta(p, q)} = \frac{1}{\beta(p, q)} \int_0^x t^{p-1}(1-t)^{q-1}dt \text{ for } 0 \leq x \leq 1, p > 0, q > 0
\]

See **Beta** for the definition of \(\beta(p, q)\).

The parameters \(p\) and \(q\) must both be greater than zero. The argument \(x\) must lie in the range 0 to 1. The incomplete beta function can underflow for sufficiently small \(x\) and large \(p\); however, this underflow is not reported as an error. Instead, the value zero is returned as the function value.

The method **BetaIncomplete** is based on the work of Bosten and Battiste (1974).

- **x** A **double** value specifying the upper limit of integration. It must be in the interval [0,1] inclusive.
- **p** A **double** value specifying the first Beta parameter. It must be positive.
- **q** A **double** value specifying the second Beta parameter. It must be positive.

Returns — A **double** value specifying the incomplete Beta function ratio.

Cot

```csharp
static public double Cot(double x)
```

Returns the cotangent of a **double**.

- **x** A **double** value

Returns — A **double** value specifying the cotangent of \(x\). If \(x\) is NaN, the result is NaN.
Erf

\texttt{static public double Erf(double x)}

Returns the error function of a double.

The error function method, \texttt{Erf(x)}, is defined to be

\[
\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt
\]

All values of \(x\) are legal.
x  A double value.

Returns — A double value specifying the error function of x.

ErfInverse

static public double ErfInverse(double x)

Returns the inverse of the error function.
ErfInverse(X) method computes the inverse of the error function erf x, defined in Erf.

The method ErfInverse(X) is defined for $x_{\max} < |x| < 1$, then the answer will be less accurate than half precision. Very approximately,

$$x_{\max} \approx 1 - \sqrt{\varepsilon/(4\pi)}$$

where $\varepsilon$ is the machine precision (approximately 1.11e-16).
`x` A double value.

Returns — A double value specifying the inverse of the error function of `x`.

**Erfc**

```csharp
static public double Erfc(double x)
```

Returns the complementary error function of a double.
The complementary error function method, \texttt{Erfc}(x), is defined to be

\[
erfc(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt
\]

The argument \( x \) must not be so large that the result underflows. Approximately, \( x \) should be less than

\[
\left[ -\ln \left( \sqrt{\pi} s \right) \right]^{1/2}
\]

where \( s = \texttt{Double.Epsilon} \) is the smallest representable positive floating-point number.
Complementary Error Function

-3.00 -2.00 -1.00 0.00 1.00 2.00 3.00

\( x \)

\( \text{erfc}(x) \)

-3.00 -2.00 -1.00 0.00 1.00 2.00 3.00

\( x \)

A double value.

Returns — A double value specifying the complementary error function of \( x \).

\text{ErfcInverse}

\text{static public double ErfcInverse(double x)}

Returns the inverse of the complementary error function.
The `ErfcInv(x)` method computes the inverse of the complementary error function erfc x, defined in `Erfc`.

`ErfcInv(x)` is defined for $0 < x < 2$. If $x_{\text{max}} < x < 2$, then the answer will be less accurate than half precision. Very approximately,

$$x_{\text{max}} \approx 2 - \sqrt{\varepsilon/(4\pi)}$$

where $\varepsilon = \text{machine precision}$ (approximately 1.11e-16).
The `Inverse Complementary Error Function` is a mathematical function that is the inverse of the complementary error function, `erfc(x)`. The function takes a `double` value, with the condition `0 ≤ x ≤ 2`, and returns a `double` value specifying the inverse of the error function of `x`.

Fact

```csharp
static public double Fact(int n)
Returns the factorial of an integer.
```
n An int value.

Returns — A double value specifying the factorial of n, n!. If x is negative, the result is NaN.

Gamma

static public double Gamma(double x)

Returns the Gamma function of a double.

The Gamma function, \( \Gamma(x) \), is defined to be

\[
\Gamma (x) = \int_{0}^{\infty} t^{x-1}e^{-t}dt \quad \text{for } x > 0
\]

For \( x < 0 \), the above definition is extended by analytic continuation.

The Gamma function is not defined for integers less than or equal to zero. Also, the argument \( x \) must be greater than \(-170.56\) so that \( \Gamma(x) \) does not underflow, and \( x \) must be less than \( 171.64 \) so that \( \Gamma(x) \) does not overflow. The underflow limit occurs first for arguments that are close to large negative half integers. Even though other arguments away from these half integers may yield machine-representable values of \( \Gamma(x) \), such arguments are considered illegal. Users who need such values should use the Log Gamma. Finally, the argument should not be so close to a negative integer that the result is less accurate than half precision.
A double value.

Returns — A double value specifying the Gamma function of x. If x is a negative integer, the result is NaN.

Log10

static public double Log10(double x)

Returns the common (base 10) logarithm of a double.
A double value.

Returns — A double value specifying the common logarithm of \( x \).

Log1p

```java
static public double Log1p(double x)
```

Returns \( \log(1+x) \), the logarithm of \( (x + 1) \).

Specifically:

- \( \log(1+0) \) returns \( +0 \).
- \( \log(1-1) \) returns \( -\infty \).
- \( \log(1-x) \) returns NaN, if \( x < -1 \).
- \( \log(1+\infty) \) returns \( +\infty \).

\( x \) A double value representing the argument.

Returns — A double value representing \( \log(1+x) \).

LogBeta

```java
static public double LogBeta(double a, double b)
```

Returns the logarithm of the Beta function.

Method LogBeta computes \( \ln \beta(a, b) = \ln \beta(b, a) \). See Beta for the definition of \( \beta(a, b) \).

LogBeta is defined for \( a \geq 0 \) and \( b \geq 0 \). It returns accurate results even when \( a \) or \( b \) is very small. It can overflow for very large arguments; this error condition is not detected except by the computer hardware.

\( a \) A double value.
\( b \) A double value.

Returns — A double value specifying the natural logarithm of the Beta function.

LogGamma

```java
static public double LogGamma(double x)
```

Returns the logarithm of the Gamma function of the absolute value of a double.

Method LogGamma computes \( \ln|\Gamma(x)| \). See Gamma for the definition of \( \Gamma(x) \).

The Gamma function is not defined for integers less than or equal to zero. Also, \(|x|\) must not be so large that the result overflows. Neither should \( x \) be so close to a negative integer that the accuracy is worse than half precision.
x  A double value.

Returns — A double value specifying the natural logarithm of the Gamma function of |x|. If x is a negative integer, the result is NaN.

Poch

    static public double Poch(double a, double x)

    Returns a generalization of Pochhammer’s symbol.
Method Poch evaluates Pochhammer’s symbol \((a)_n = (a)(a-1)\ldots(a-n+1)\) for \(n\) a nonnegative integer. Pochhammer’s generalized symbol is defined to be

\[ (a)_x = \frac{\Gamma(a+x)}{\Gamma(a)} \]

See Gamma for the definition of \(\Gamma(x)\).

Note that a straightforward evaluation of Pochhammer’s generalized symbol with either Gamma or Log Gamma functions can be especially unreliable when \(a\) is large or \(x\) is small.

Substantial loss can occur if \(a + x\) or \(a\) are close to a negative integer unless \(|x|\) is sufficiently small. To insure that the result does not overflow or underflow, one can keep the arguments \(a\) and \(a + x\) well within the range dictated by the Gamma function method Gamma or one can keep \(|x|\) small whenever \(a\) is large. Poch also works for a variety of arguments outside these rough limits, but any more general limits that are also useful are difficult to specify.

\(a\) — A double value specifying the first argument.
\(x\) — A double value specifying the second, differential argument.

Returns — A double value specifying the generalized Pochhammer symbol, \(\Gamma(a+x)/\Gamma(a)\).

R9lgmc

\textbf{static public double R9lgmc(double x)}

Returns the Log Gamma correction term for argument values greater than or equal to 10.0.

\(x\) — A double value.

Returns — A double value specifying the Log Gamma correction term.

Sign

\textbf{static public double Sign(double x, double y)}

Returns the value of \(x\) with the sign of \(y\).

\(x\) — A double value.
\(y\) — A double value.

Returns — A double value specifying the absolute value of \(x\) and the sign of \(y\).

\textbf{Example: The Special Functions}

Various special functions are exercised. Their use in this example typifies the manner in which other special functions in the Sfun class would be used.
using System;
using Imsl.Math;

class SfunEx1
{
    public static void Main(string[] args)
    {
        double result;

        // Log base 10 of x
        double x = 100.0;
        result = Sfun.Log10(x);
        Console.WriteLine("The log base 10 of 100 is "+ result);

        // Factorial of 10
        int n = 10;
        result = Sfun.Fact(n);
        Console.WriteLine("10 factorial is "+ result);

        // Gamma of 5.0
        double x1 = 5.0;
        result = Sfun.Gamma(x1);
        Console.WriteLine("The Gamma function at 5.0 is "+ result);

        // LogGamma of 1.85
        double x2 = 1.85;
        result = Sfun.LogGamma(x2);
        Console.WriteLine("The logarithm of the absolute value of the Gamma function 

                                 at 1.85 is "+ result);

        // Beta of (2.2, 3.7)
        double a = 2.2;
        double b = 3.7;
        result = Sfun.Beta(a, b);
        Console.WriteLine("Beta(2.2, 3.7) is "+ result);

        // LogBeta of (2.2, 3.7)
        double a1 = 2.2;
        double b1 = 3.7;
        result = Sfun.LogBeta(a1, b1);
        Console.WriteLine("logBeta(2.2, 3.7) is "+ result + "\n");
    }
}

Output

The log base 10 of 100 is 2
10 factorial is 3628800
The Gamma function at 5.0 is 24
The logarithm of the absolute value of the Gamma function
at 1.85 is -0.0559238130196572
Bessel Class

Collection of Bessel functions.

public class Imsl.Math.Bessel

Methods

\texttt{I}

\texttt{static public double[]} \texttt{I(double x, int n)}

Evaluates a sequence of modified Bessel functions of the first kind with integer order and real argument.

Bessel.I[i] contains the value of the Bessel function of order \( i \). The Bessel function \( I_n(x) \) is defined to be

\[
I_n(x) = \frac{1}{\pi} \int_0^\pi e^{x \cos \theta} \cos(n \theta) \, d\theta
\]

The input \( x \) must satisfy \(|x| \leq \log(b)\) where \( b \) is the largest representable floating-point number. The algorithm is based on a code due to Sookne (1973b), which uses backward recursion.

\( x \) A double representing the argument of the Bessel functions to be evaluated.

\( n \) The int order of the last element in the sequence.

Returns — A double array of length \( n+1 \) containing the values of the function through the series.

\texttt{I}

\texttt{static public double[]} \texttt{I(double xnu, double x, int n)}

Evaluates a sequence of modified Bessel functions of the first kind with real order and real argument.

Bessel.I[i] contains the value of the Bessel function of order \( i+xnu \). The Bessel function \( I_\nu(x) \), is defined to be

\[
I_\nu(x) = \frac{1}{\pi} \int_0^\pi e^{x \cos \theta} \cos(\nu \theta) \, d\theta - \frac{\sin(\nu \pi)}{\nu \pi} \int_0^\infty e^{-x \cosh t - \nu t} \, dt
\]

Here, argument \( xnu \) is represented by \( \nu \) in the above equation.
The input \( x \) must be nonnegative and less than or equal to \( \log(b) \) (\( b \) is the largest representable number). The argument \( \nu = x_{\text{nu}} \) must satisfy \( 0 \leq \nu \leq 1 \).

This function is based on a code due to Cody (1983), which uses backward recursion.

**xnu**: A double representing the lowest order desired. \( x_{\text{nu}} \) must be at least zero and less than 1.

**x**: A double representing the argument of the Bessel functions to be evaluated.

**n**: The int order of the last element in the sequence.

**Returns**: A double array of length \( n + 1 \) containing the values of the function through the series.

\[
\text{J static public double[] J(double x, int n)}
\]

Evaluates a sequence of Bessel functions of the first kind with integer order and real argument.

\( \text{Bessel.J}[i] \) contains the value of the Bessel function of order \( i \) at \( x \) for \( i = 0 \) to \( n \). The Bessel function \( J_n(x) \), is defined to be

\[
J_n(x) = \frac{1}{\pi} \int_0^\pi \cos(x \sin \theta - n \theta) \, d\theta
\]

The algorithm is based on a code due to Sookne (1973b) that uses backward recursion with strict error control.

**x**: A double representing the argument for which the sequence of Bessel functions is to be evaluated.

**n**: A int which specifies the order of the last element in the sequence.

**Returns**: A double array of length \( n + 1 \) containing the values of the function through the series.

\[
\text{J static public double[] J(double xnu, double x, int n)}
\]

Evaluate a sequence of Bessel functions of the first kind with real order and real positive argument.

The Bessel function \( J_\nu(x) \), is defined to be

\[
J_\nu(x) = \frac{(x/2)^\nu}{\sqrt{\pi} \Gamma(\nu + 1/2)} \int_0^\pi \cos(x \cos \theta) \sin^{2\nu} \theta \, d\theta
\]

This code is based on the work of Gautschi (1964) and Skovgaard (1975). It uses backward recursion.

**xnu**: A double representing the lowest order desired. \( x_{\text{nu}} \) must be at least zero and less than 1.

**x**: A double representing the argument for which the sequence of Bessel functions is to be evaluated.
n  A \textbf{int} representing the order of the last element in the sequence. If order is the
highest order desired, set \textit{n} to \textbf{int}(order).

Returns — A \textbf{double} array of length \textit{n}+1 containing the values of the function through
the series. Bessel.J[\textit{i}] contains the value of the Bessel function of order \textit{i} + \nu at \textit{x} for \textit{i}=0
to \textit{n}.

K

\textbf{static} public \textbf{double}[\textit{i}] \textbf{K}(\textbf{double} \textit{x}, \textbf{int} \textit{n})

Evaluates a sequence of modified Bessel functions of the third kind with integer order and
real argument.

This function uses $e^xK_{\nu+k-1}$ for \textit{k} = 1, \ldots, \textit{n} and \nu = 0. For the definition of $K_{\nu}(x)$, see
below.

\textbf{x}  A \textbf{double} representing the argument for which the sequence of Bessel functions is to
be evaluated.

\textbf{n}  A \textbf{int} which specifies the order of the last element in the sequence.

Returns — A \textbf{double} array of length \textit{n} + 1 containing the values of the function through
the series.

K

\textbf{static} public \textbf{double}[\textit{i}] \textbf{K}(\textbf{double} \textit{xnu}, \textbf{double} \textit{x}, \textbf{int} \textit{n})

Evaluates a sequence of modified Bessel functions of the third kind with fractional order
and real argument.

Bessel.K[\textit{i}] contains the value of the Bessel function of order \textit{i} + \nu at \textit{x} for \textit{i}=0 to \textit{n}.

The Bessel function $K_{\nu}(x)$ is defined to be

$$K_{\nu}(x) = \frac{\pi}{2} e^{\nu \pi i/2} \left[ i J_{\nu}(ix) - Y_{\nu}(ix) \right] \quad \text{for } -\pi < \arg x \leq \frac{\pi}{2}$$

Currently, \textit{xnu} (represented by \nu in the above equation) is restricted to be less than one
in absolute value. A total of \textit{n} values is stored in the result, \textbf{K}.

\textbf{K}[0] = $K_{\nu}(x)$, \textbf{K}[1] = $K_{\nu+1}(x)$, \ldots, \textbf{K} [\textit{n} - 1] = $K_{\nu+n-1}(x)$.

This method is based on the work of Cody (1983).

\textbf{xnu}  A \textbf{double} representing the fractional order of the function. \textbf{xnu} must be less than
one in absolute value.

\textbf{x}  A \textbf{double} representing the argument for which the sequence of Bessel functions is to
be evaluated.

\textbf{n}  A \textbf{int} representing the order of the last element in the sequence. If order is the
highest order desired, set \textbf{n} to \textbf{int}(order).

Returns — A \textbf{double} array of length \textit{n}+1 containing the values of the function through
the series.
ScaledK

static public double[] ScaledK(double v, double x, int n)
Evaluate a sequence of exponentially scaled modified Bessel functions of the third kind with fractional order and real argument.

If \( n \) is positive, \( \text{Bessel.K}[I] \) contains \( e^x \) times the value of the Bessel function of order \( I + v \) at \( x \) for \( I = 0 \) to \( n \).

If \( n \) is negative, \( \text{Bessel.K}[I] \) contains \( e^x \) times the value of the Bessel function of order \( v - I \) at \( x \) for \( I = 0 \) to \( n \). This function evaluates \( e^x K_{\nu+n-1}(x) \), for \( i=1,...,n \) where \( K \) is the modified Bessel function of the third kind. Currently, \( v \) is restricted to be less than 1 in absolute value. A total of \(|n| + 1\) elements are returned in the array. This code is particularly useful for calculating sequences for large \( x \) provided \( n = x \). (Overflow becomes a problem if \( n << x \).) \( n \) must not be zero, and \( x \) must be greater than zero. \(|\nu| \) must be less than 1. Also, when \(|n| \) is large compared with \( x \), \(|\nu+n| \) must not be so large that \( e^x K_{\nu+n}(x) \) overflows. The code is based on work of Cody (1983).

\( v \) A double representing the fractional order of the function. \( v \) must be less than one in absolute value.

\( x \) A double representing the argument for which the sequence of Bessel functions is to be evaluated.

\( n \) A int representing the order of the last element in the sequence. If order is the highest order desired, set \( n \) to int(order).

Returns — A double array of length \( n+1 \) containing the values of the function through the series.

Y

static public double[] Y(double xnu, double x, int n)
Evaluate a sequence of Bessel functions of the second kind with real nonnegative order and real positive argument.

\( \text{Bessel.K}[I] \) contains the value of the Bessel function of order \( I + v \) at \( x \) for \( I = 0 \) to \( n \). The Bessel function \( Y_\nu(x) \) is defined to be

\[
Y_\nu(x) = \frac{1}{\pi} \int_0^\pi \cos(x \sin \theta - \nu \theta) d\theta
\]

\[
-\frac{1}{\pi} \int_0^\infty \left[ e^{\nu t} + e^{-\nu t} \cos(\nu \pi) \right] e^{-x \sinh t} dt
\]

The variable \( xnu \) (represented by \( \nu \) in the above equation) must satisfy \( 0 \leq \nu < 1 \). If this condition is not met, then \( Y \) is set to NaN. In addition, \( x \) must be in \([x_m, x_M]\) where
\[ x_m = 6(16^{-32}) \text{ and } x_m = 16^9. \] If \( x < x_m \), then the largest representable number is returned; and if \( x < x_M \), then zero is returned.

The algorithm is based on work of Cody and others, (see Cody et al. 1976; Cody 1969; NATS FUNPACK 1976). It uses a special series expansion for small arguments. For moderate arguments, an analytic continuation in the argument based on Taylor series with special rational minimax approximations providing starting values is employed. An asymptotic expansion is used for large arguments.

**xnu** A **double** representing the lowest order desired. **xnu** must be at least zero and less than 1.

**x** A **double** representing the argument for which the sequence of Bessel functions is to be evaluated.

**n** A **int** which specifies that \( n + 1 \) elements will be evaluated in the sequence.

Returns — A **double** array of length \( n + 1 \) containing the values of the function through the series.

**Example: The Bessel Functions**

The Bessel functions I, J, and K are exercised for orders 0, 1, 2, and 3 at argument 10.e0.

```csharp
using System;
using Imsl.Math;

public class BesselEx1
{
    public static void Main(String[] args)
    {
        double x = 10e0;
        int hiorder = 4;
        // Exercise some of the Bessel functions with argument 10.0
        double[] bi = Bessel.I(x, hiorder);
        double[] bj = Bessel.J(x, hiorder);
        double[] bk = Bessel.K(x, hiorder);
        for (int i = 0; i < 4; i++)
        {
            Console.Out.WriteLine(i + " " + bi[i] + " " + bj[i] + " " + bk[i]);
        }
        Console.Out.WriteLine();
    }
}
```

**Output**

<table>
<thead>
<tr>
<th>Order</th>
<th>Bessel.I</th>
<th>Bessel.J</th>
<th>Bessel.K</th>
</tr>
</thead>
</table>

**Special Functions**

Bessel Class • 191
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2815.71662846626</td>
<td>-0.245935764451348</td>
<td>1.7780623161676E-05</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2670.98830370126</td>
<td>0.0434727461688615</td>
<td>1.86487734538256E-05</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2281.51896772601</td>
<td>0.254630313685121</td>
<td>2.15098170069328E-05</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1758.38071661085</td>
<td>0.0583793793051867</td>
<td>2.72527002565987E-05</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 10: Miscellaneous

Types

structure Complex ................................................................. 193
structure Physical ................................................................. 211
class EpsilonAlgorithm ......................................................... 222

Complex Structure

Set of mathematical functions for complex numbers. It provides the basic operations (addition, subtraction, multiplication, division) as well as a set of complex functions.

public structure Imsl.Math.Complex implements Field

I

public Imsl.Math.Complex I
The imaginary unit. This constant is set to new Complex(0,1).

Constructors

Complex

public Complex(Imsl.Math.Complex z)
Constructs a Complex equal to the argument.

z A Complex object.

System.NullReferenceException is thrown if z is null
public Complex(double re, double im)
Constructs a Complex with real and imaginary parts given by the input arguments.

- **re** A double value equal to the real part of the Complex object.
- **im** A double value equal to the imaginary part of the Complex object.

public Complex(double re)
Constructs a Complex with a zero imaginary part.

- **re** A double value equal to the real part of the Complex object.

**Operators**

+ static public operator x + y
Returns the sum of two Complex objects, x+y.

- **x** A Complex object.
- **y** A Complex object.

Returns — A newly constructed Complex initialized to x+y.

+ static public operator x + y
Returns the sum of a Complex and a double, x+y.

- **x** A Complex object.
- **y** A double value.

Returns — A newly constructed Complex initialized to x+y.

+ static public operator x + y
Returns the sum of a double and a Complex, x+y.

- **x** A double value.
- **y** A Complex object.

Returns — A newly constructed Complex initialized to x+y.

/ static public operator x / y
Returns the result of a Complex object divided by a Complex object, x/y.

- **x** A Complex object representing the numerator.
- **y** A Complex object representing the denominator.
Returns — A newly constructed Complex initialized to x/y.

static public operator x / y
Returns the result of a Complex object divided by a double, x/y.

x  A Complex object representing the numerator.
y  A double representing the denominator.

Returns — A newly constructed Complex initialized to x/y.

static public operator x / y
Returns the result of a double divided by a Complex object, x/y.

x  A double value.
y  A Complex object representing the denominator.

Returns — A newly constructed Complex initialized to x/y.

==
static public operator x == y
Returns true if x and y are equal.

x  A Complex value.
y  A Complex value.

Returns — true if x and y are equal.

==
static public operator x == y
Returns true if x and y are equal.

x  A Complex value.
y  A double value.

Returns — true if x and y are equal.

==
static public operator x == y
Returns true if x and y are equal.

x  A double value.
y  A Complex value.

Returns — true if x and y are equal.

!=
static public operator x != y
Returns true if x and y are not equal.
x  A Complex value.
y  A double value.
Returns — true if x and y are equal.
!
static public operator x != y
Returns true if x and y are not equal.
x  A double value.
y  A Complex value.
Returns — true if x and y are not equal.
!
static public operator x != y
Returns true if x and y are not equal.
x  A Complex value.
y  A Complex value.
Returns — true if x and y are equal.
*
static public operator x * y
Returns the product of two Complex objects, x * y.
x  A Complex object.
y  A Complex object.
Returns — A newly constructed Complex initialized to x * y.
*
static public operator x * y
Returns the product of a Complex object and a double, x * y.
x  A Complex object.
y  A double value.
Returns — A newly constructed Complex initialized to x * y.
*
static public operator x * y
Returns the product of a double and a Complex object, x * y.
x  A double value.
y  A Complex object.
Returns — A newly constructed Complex initialized to x * y.
- static public operator x - y
Returns the difference of two Complex objects, x-y.

x  A Complex object.
y  A Complex object.
Returns — A newly constructed Complex initialized to x-y.

- static public operator x - y
Returns the difference of a Complex object and a double, x-y.

x  A Complex object.
y  A double value.
Returns — A newly constructed Complex initialized to x-y.

- static public operator x - y
Returns the difference of a double and a Complex object, x-y.

x  A double value.
y  A Complex object.
Returns — A newly constructed Complex initialized to x-y.

- static public operator - x
Returns the negative of a Complex object, -x.

x  A Complex object.
Returns — A newly constructed Complex initialized to the negative of the Complex argument, x.

Methods

Abs
static public double Abs(Imsl.Math.Complex z)
Returns the absolute value (modulus) of a Complex, |z|.

z  A Complex object.
Returns — A double value equal to the absolute value of the argument.

Acos
Returns the inverse cosine (arc cosine) of a Complex, with branch cuts outside the interval [-1,1] along the real axis.
Specifically, if \( z = x + iy \),
\[
\text{acos}(\bar{z}) = \text{acos}(z).
\]
\[
\text{acos}(\pm 0 + i0) \text{ returns } \pi/2 - i0.
\]
\[
\text{acos}(\pm \infty + i\infty) \text{ returns } \pi/4 - i\infty.
\]
\[
\text{acos}(x + i\infty) \text{ returns } \pi/2 - i\infty, \text{ for finite } x.
\]
\[
\text{acos}(-\infty + iy) \text{ returns } \pi - i\infty, \text{ for positive-signed finite } y.
\]
\[
\text{acos}(\pm \infty + iy) \text{ returns } +0 - i\infty, \text{ for positive-signed finite } y.
\]
\[
\text{acos}(\pm \infty + i\text{NaN}) \text{ returns } \text{NaN} \pm i\infty \text{ (where the sign of the imaginary part of the result is unspecified)}.
\]
\[
\text{acos}(\pm 0 + i\text{NaN}) \text{ returns } \pi/2 + i\text{NaN}.
\]
\[
\text{acos}(\text{NaN} + i\infty) \text{ returns } \text{NaN} - i\infty.
\]
\[
\text{acos}(x + i\text{NaN}) \text{ returns } \text{NaN} + i\text{NaN}, \text{ for nonzero finite } x.
\]
\[
\text{acos}(\text{NaN} + iy) \text{ returns } \text{NaN} + i\text{NaN}, \text{ for finite } y.
\]
\[
\text{acos}(\text{NaN} + i\text{NaN}) \text{ returns } \text{NaN} + i\text{NaN}.
\]

\( z \) A Complex object.

Returns — A newly constructed Complex initialized to the inverse (arc) cosine of the argument. The real part of the result is in the interval \([0, \pi]\).

**Acosh**

```csharp
```

Returns the inverse hyperbolic cosine (arc cosh) of a Complex, with a branch cut at values less than one along the real axis.

Specifically, if \( z = x + iy \),
\[
\text{acosh}(\bar{z}) = \text{acosh}(z).
\]
\[
\text{acosh}(\pm 0 + i0) \text{ returns } +0 + i\pi/2.
\]
\[
\text{acosh}(\pm \infty + i\infty) \text{ returns } +\infty + i3\pi/4.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]
\[
\text{acosh}(\pm \infty + i\text{NaN}) \text{ returns } +\infty + i\text{NaN}.
\]

\( z \) A Complex object.
Returns — A newly constructed Complex initialized to the inverse (arc) hyperbolic cosine of the argument. The real part of the result is non-negative and its imaginary part is in the interval \([-i\pi, i\pi]\).

**Add**

```java
```

Returns the sum of two Complex objects, \(x+y\).

- **x** A Complex object.
- **y** A Complex object.

Returns — A newly constructed Complex initialized to \(x+y\).

**Add**

```java
```

Returns the sum of a Complex and a double, \(x+y\).

- **x** A Complex object.
- **y** A double value.

Returns — A newly constructed Complex initialized to \(x+y\).

**Add**

```java
static public Imsl.Math.Complex Add(double x, Imsl.Math.Complex y)
```

Returns the sum of a double and a Complex, \(x+y\).

- **x** A double value.
- **y** A Complex object.

Returns — A newly constructed Complex initialized to \(x+y\).

**Argument**

```java
static public double Argument(Imsl.Math.Complex z)
```

Returns the argument (phase) of a Complex, in radians, with a branch cut along the negative real axis.

- **z** A Complex object.

Returns — A double value equal to the argument (or phase) of a Complex. It is in the interval \([-\pi, \pi]\).

**Asin**

```java
```

Returns the inverse sine (arc sine) of a Complex, with branch cuts outside the interval \([-1,1]\) along the real axis.

The value of Asin is defined in terms of the function Asinh, by \(\text{asin}(z) = -i \text{asinh}(iz)\).


- **z** A Complex object.
Returns — A newly constructed Complex initialized to the inverse (arc) sine of the argument. The real part of the result is in the interval $[-\pi/2, +\pi/2]$.

**Asinh**

```csharp
```

Returns the inverse hyperbolic sine (arc sinh) of a Complex, with branch cuts outside the interval $[-i, i]$.

Specifically, if $z = x + iy$,

- $\text{asinh}(\bar{z}) = \text{asinh}(\bar{z})$ and asinh is odd.
- $\text{asinh}(0 + i0)$ returns $0 + i0$.
- $\text{asinh}(\infty + i\infty)$ returns $+\infty + i\pi/4$.
- $\text{asinh}(x + i\infty)$ returns $+\infty + i\pi/2$ for positive-signed finite $x$.
- $\text{asinh}(+0 + i0)$ returns $+0 + i0$.
- $\text{asinh}(+\infty + iy)$ returns $+\infty + i0$ for positive-signed finite $y$.
- $\text{asinh}(\text{NaN} + i\infty)$ returns $\pm\infty + i\text{NaN}$ (where the sign of the real part of the result is unspecified).
- $\text{asinh}(+\infty + i\text{NaN})$ returns $+\infty + i\text{NaN}$.
- $\text{asinh}(\text{NaN} + i0)$ returns $\text{NaN} + i0$.
- $\text{asinh}(\text{NaN} + iy)$ returns $\text{NaN} + i\text{NaN}$, for finite nonzero $y$.
- $\text{asinh}(x + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$, for finite $x$.
- $\text{asinh}(\text{NaN} + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$.

$z$ A Complex object.

Returns — A newly constructed Complex initialized to the inverse (arc) hyperbolic sine of the argument. Its imaginary part is in the interval $[-i\pi/2, i\pi/2]$.

**Atan**

```csharp
```

Returns the inverse tangent (arc tangent) of a Complex, with branch cuts outside the interval $[-i, i]$ along the imaginary axis.

The value of Atan is defined in terms of the function Atanh, by $\text{atan}(z) = -i\text{atanh}(iz)$.


$z$ A Complex object.

Returns — A newly constructed Complex initialized to the inverse (arc) tangent of the argument. Its real part is in the interval $[-\pi/2, \pi/2]$.

**Atanh**

```csharp
```

Returns the inverse hyperbolic tangent (arc tanh) of a Complex, with branch cuts outside the interval $[-1, 1]$ on the real axis.

Specifically, if $z = x + iy$,

- $\text{atanh}(\bar{z}) = \text{atanh}(\bar{z})$ and atanh is odd.
atanh(+0 + i0) returns +0 + i0.
atanh(+∞ + i∞) returns +0 + iπ/2.
atanh(+∞ + iy) returns +0 + iπ/2, for finite positive-signed y.
atanh(x + i∞) returns +0 + iπ/2, for finite positive-signed x.
atanh(+0 + iNaN) returns +0 + iNaN.
atanh(NaN + i∞) returns ±0 + iπ/2 (where the sign of the real part of the result is unspecified).
atanh(+∞ + iNaN) returns +0 + iNaN.
atanh(NaN + iy) returns NaN + iNaN, for finite y.
atanh(x + iNaN) returns NaN + iNaN, for nonzero finite x.

z  A Complex object.

Returns — A newly constructed Complex initialized to the inverse (arc) hyperbolic tangent of the argument. The imaginary part of the result is in the interval [−iπ/2, iπ/2].

CompareTo

Final public int CompareTo(Object obj)
Compares this Complex to another Object.

If the Object is a Complex, this function behaves like compareTo(Complex). Otherwise, it throws a InvalidCastException (as Complex objects are comparable only to other Complex objects).

obj  An Object to be compared.

Returns — An int, 0 if obj is equal to this Complex; a value less than 0 if this Complex is less than obj; and a value greater than 0 if this Complex is greater than obj.

System.InvalidCastException is thrown if obj is not a Complex object

CompareTo

public int CompareTo(Imsl.Math.Complex z)
Compares two Complex objects.

A lexicographical ordering is used. First the real parts are compared in the sense of Double.compareTo. If the real parts are unequal this is the return value. If the return parts are equal then the comparison of the imaginary parts is returned.

z  A Complex to be compared.

Returns — The value 0 if z is equal to this Complex; a value less than 0 if this Complex is less than z; and a value greater than 0 if this Complex is greater than z.

Conjugate

Returns the complex conjugate of a Complex object.
z  A Complex object.

Returns — A newly constructed Complex initialized to the complex conjugate of Complex argument, z.

Cos
Returns the cosine of a Complex.
The value of Cos is defined in terms of the function Cosh, by \( \cos(z) = \cosh(iz) \).

z  A Complex object.

Returns — A newly constructed Complex initialized to the cosine of the argument.

Cosh
Returns the hyperbolic cosh of a Complex.

If \( z = x + iy \),
cosh(\bar{z}) = \cosh(z) and cosh is even.
cosh(+0 + i0) returns 1 + i0.
cosh(+0 + i\infty) returns NaN ± i0 (where the sign of the imaginary part of the result is unspecified).
cosh(+\infty + i0) returns +\infty + i0.
cosh(+\infty + i\infty) returns +\infty + iNaN.
cosh(x + i\infty) returns NaN + iNaN, for finite nonzero x.
cosh(+\infty + iy) returns +\infty[\cos(y) + i\sin(y)], for finite nonzero y.
cosh(+0 + iNaN) returns NaN ± i0 (where the sign of the imaginary part of the result is unspecified).
cosh(+\infty + iNaN) returns +\infty + iNaN.
cosh(x + iNaN) returns NaN + iNaN, for finite nonzero x.
cosh(NaN + i0) returns NaN ± i0 (where the sign of the imaginary part of the result is unspecified).
cosh(NaN + iy) returns NaN + iNaN, for all nonzero numbers y.
cosh(NaN + iNaN) returns NaN + iNaN.

z  A Complex object.

Returns — A newly constructed Complex initialized to the hyperbolic cosine of the argument.

Divide
Returns the result of a Complex object divided by a Complex object, \( x/y \).
x A Complex object representing the numerator.
y A Complex object representing the denominator.
Returns — A newly constructed Complex initialized to x/y.

Divide
Returns the result of a Complex object divided by a double, x/y.

x A Complex object representing the numerator.
y A double representing the denominator.
Returns — A newly constructed Complex initialized to x/y.

Divide
Returns the result of a double divided by a Complex object, x/y.

x A double value.
y A Complex object representing the denominator.
Returns — A newly constructed Complex initialized to x/y.

Equals
override public bool Equals(Object x)
Compares this object against the specified object.

x The object to compare with.
Returns — true if the objects are the same; false otherwise.

Exp
Returns the exponential of a Complex z, exp(z).
Specifically, if z = x+iy,
exp(z) = exp(z).
exp(±0 + i0) returns 1 + i0.
exp(+∞ + i0) returns +∞ + i0.
exp(−∞ + i∞) returns ±0 ± i0 (where the signs of the real and imaginary parts of the
result are unspecified).
exp(+∞ + i∞) returns ±∞ + iNaN (where the sign of the real part of the result is
unspecified).
exp(x + i∞) returns NaN + iNaN, for finite x.
exp(−∞ + iy) returns +0[cos(y) + i sin(y)], for finite y.
exp(+∞ + iy) returns +∞[cos(y) + i sin(y)], for finite nonzero y.
exp(−∞ + iNaN) returns ±0 ± i0 (where the signs of the real and imaginary parts of the
result are unspecified).
exp(\(+\infty + iNaN\)) returns \(\pm\infty + iNaN\) (where the sign of the real part of the result is unspecified).
exp(NaN + i0) returns NaN + i0.
exp(NaN + iy) returns NaN + iNaN, for all non-zero numbers \(y\).
exp(\(x + iNaN\)) returns NaN + iNaN, for finite \(x\).

\(z\) A Complex object.

Returns — A newly constructed Complex initialized to the exponential of the argument.

GetHashCode

```csharp
override public int GetHashCode()
Returns a hashcode for this Complex.

Returns — A hash code value for this object.
```

Imag

```csharp
public double Imag()
Returns the imaginary part of a Complex object.

Returns — A double representing the imaginary part of a Complex object, \(z\).
```

Imag

```csharp
static public double Imag(Imsl.Math.Complex \(z\))
Returns the imaginary part of a Complex object.

\(z\) A Complex object.

Returns — A double representing the imaginary part of the Complex object, \(z\).
```

Log

```csharp
static public Imsl.Math.Complex Log(Imsl.Math.Complex \(z\))
Returns the logarithm of a Complex \(z\), with a branch cut along the negative real axis.

Specifically, if \(z = x+iy\),
log(\(\bar{z}\)) = \(\overline{\log(z)}\).
log(0 + i0) returns \(-\infty + i\pi\).
log(+0 + i0) returns \(-\infty + i0\).
log(-\(\infty + i\infty\)) returns \(+\infty + i3\pi/4\).
log(+\(\infty + i\infty\)) returns \(+\infty + i\pi/4\).
log(\(x + i\infty\)) returns \(+\infty + i\pi/2\), for finite \(x\).
log(-\(\infty + iy\)) returns \(+\infty + i\pi\), for finite positive-signed \(y\).
log(+\(\infty + iy\)) returns \(+\infty + i0\), for finite positive-signed \(y\).
log(\(\pm\infty + iNaN\)) returns \(+\infty + iNaN\).
log(NaN + i\(\infty\)) returns \(+\infty + iNaN\).
log(\(x + iNaN\)) returns NaN + iNaN, for finite \(x\).
```
log(NaN + iy) returns NaN + iNaN, for finite y.
log(NaN + iNaN) returns NaN + iNaN.

z A Complex object.

Returns — A newly constructed Complex initialized to the logarithm of the argument. Its imaginary part is in the interval [−iπ, iπ].

Multiply
Returns the product of two Complex objects, x * y.

x A Complex object.
y A Complex object.

Returns — A newly constructed Complex initialized to x × y.

Multiply
Returns the product of a Complex object and a double, x * y.

x A Complex object.
y A double value.

Returns — A newly constructed Complex initialized to x × y.

Multiply
static public Imsl.Math.Complex Multiply(double x, Imsl.Math.Complex y)
Returns the product of a double and a Complex object, x * y.

x A double value.
y A Complex object.

Returns — A newly constructed Complex initialized to x × y.

MultiplyImag
Returns the product of a Complex object and a pure imaginary double, x * iy.

x A Complex object.
y A double value representing a pure imaginary.

Returns — A newly constructed Complex initialized to x * iy.

MultiplyImag
Returns the product of a pure imaginary double and a Complex object, ix * y.

x A double value representing a pure imaginary.
A Complex object.
Returns — A newly constructed Complex initialized to \( ix \ast y \).

**Negate**

```csharp
```

Returns the negative of a Complex object, \(-z\).

\( z \)  A Complex object.

Returns — A newly constructed Complex initialized to the negative of the Complex argument, \( z \).

**Parse**

```csharp
static public Imsl.Math.Complex Parse(string s)
```

Converts the string representation of a number in a specified style to its Complex number equivalent.

\( s \)  A string containing a number to convert.

Returns — A Complex number represented in the String.

**Parse**

```csharp
static public Imsl.Math.Complex Parse(string s, System.IFormatProvider formatProvider)
```

Converts the string representation of a number in a specified style to its Complex number equivalent.

\( s \)  A string containing a number to convert.

\( formatProvider \)  An IFormatProvider that supplies culture-specific formatting information about \( s \).

Returns — A Complex number represented in the String.

**Pow**

```csharp
static public Imsl.Math.Complex Pow(Imsl.Math.Complex z, double x)
```

Returns the Complex \( z \) raised to the \( x \) power, with a branch cut for the first parameter (\( z \)) along the negative real axis.

\( z \)  A Complex object.

\( x \)  A double value.

Returns — A newly constructed Complex initialized to \( z \) to the power \( x \).

**Pow**

```csharp
```

Returns the Complex \( x \) raised to the Complex \( y \) power.

The value of Pow is defined in terms of the functions Exp and Log, by
\[
pow(x, y) = \exp(y \log(x)).
\]

x  A Complex object.
y  A Complex object.

Returns — A newly constructed Complex initialized to x^y.

Real
public double Real()
Returns the real part of a Complex object.

Returns — A double representing the real part of a Complex object, z.

Real
static public double Real(Imsl.Math.Complex z)
Returns the real part of a Complex object.

z  A Complex object.

Returns — A double representing the real part of the Complex object, z.

Sin
Returns the sine of a Complex.

The value of Sin is defined in terms of the function Sinh, by \( \sin(z) = -i\sinh(iz) \).


z  A Complex object.

Returns — A newly constructed Complex initialized to the sine of the argument.

Sinh
Returns the hyperbolic sine of a Complex.

If \( z = x+iy \),
\[
\sinh(z) = \sinh(x + iy) = \sinh(x) \cos(y) + i \sin(x) \sinh(y),
\]
for positive finite \( y \).

sinh(+\infty) returns ±\infty + iNaN (where the sign of the real part of the result is unspecified).

sinh(+0 + i\infty) returns +0 + i0.

sinh(+0 + i\infty) returns ±0 + iNaN (where the sign of the real part of the result is unspecified).

sinh(+\infty + i0) returns +\infty + i0.

sinh(+\infty + i\infty) returns ±\infty + iNaN (where the sign of the real part of the result is unspecified).

sinh(+\infty + iy) returns +\infty[\cos(y) + i \sin(y)], for positive finite \( y \).

sinh(x + i\infty) returns NaN + iNaN, for positive finite \( x \).

sinh(+0 + iNaN) returns ±0 + iNaN (where the sign of the real part of the result is unspecified).

sinh(+\infty + iNaN) returns ±\infty + iNaN (where the sign of the real part of the result is unspecified).
\[
\sinh(x + i\text{NaN}) \text{ returns } \text{NaN} + i\text{NaN}, \text{ for finite nonzero } x.
\]
\[
\sinh(\text{NaN} + i0) \text{ returns } \text{NaN} + i0.
\]
\[
\sinh(\text{NaN} + iy) \text{ returns } \text{NaN} + i\text{NaN}, \text{ for all nonzero numbers } y.
\]
\[
\sinh(\text{NaN} + i\text{NaN}) \text{ returns } \text{NaN} + i\text{NaN}.
\]

\text{z } \text{A Complex object.}

\text{Returns — A newly constructed Complex initialized to the hyperbolic sine of the argument.}

\text{Sqrt}

\text{static public Imsl.Math.Complex Sqrt(Imsl.Math.Complex z)}

\text{Returns the square root of a Complex, with a branch cut along the negative real axis.}

\text{Specifically, if } z = x+iy,
\[
\sqrt{z} = \sqrt{z}.
\]
\[
\sqrt{\pm 0 + i0} \text{ returns } +0 + i0.
\]
\[
\sqrt{(-\infty + iy)} \text{ returns } +0 + i\infty, \text{ for finite positive-signed } y.
\]
\[
\sqrt{(+\infty + iy)} \text{ returns } +\infty + i0, \text{ for finite positive-signed } y.
\]
\[
\sqrt{(x + i\infty)} \text{ returns } +\infty + i\infty, \text{ for all } x \text{ (including NaN).}
\]
\[
\sqrt{(-\infty + i\text{NaN})} \text{ returns } \text{NaN} \pm i\infty (\text{where the sign of the imaginary part of the result is unspecified}).
\]
\[
\sqrt{(+\infty + i\text{NaN})} \text{ returns } +\infty + i\text{NaN}.
\]
\[
\sqrt{(x + i\text{NaN})} \text{ returns } \text{NaN} + i\text{NaN} \text{ and optionally raises the invalid exception, for finite } x.
\]
\[
\sqrt{(\text{NaN} + iy)} \text{ returns } \text{NaN} + i\text{NaN} \text{ and optionally raises the invalid exception, for finite } y.
\]
\[
\sqrt{(\text{NaN} + i\text{NaN})} \text{ returns } \text{NaN} + i\text{NaN}.
\]

\text{z } \text{A Complex object.}

\text{Returns — A newly constructed Complex initialized to square root of } z.

\text{Subtract}


\text{Returns the difference of two Complex objects, } x-y.

\text{x } \text{A Complex object.}

\text{y } \text{A Complex object.}

\text{Returns — A newly constructed Complex initialized to } x-y.

\text{Subtract}

\text{static public Imsl.Math.Complex Subtract(Imsl.Math.Complex x, double y)}

\text{Returns the difference of a Complex object and a double, } x-y.
x A Complex object.
y A double value.

Returns — A newly constructed Complex initialized to $x - y$.

Subtract

```csharp
static public Imsl.Math.Complex Subtract(double x, Imsl.Math.Complex y)
```

Returns the difference of a double and a Complex object, $x - y$.

x A double value.
y A Complex object.

Returns — A newly constructed Complex initialized to $x - y$.

Tan

```csharp
```

Returns the tangent of a Complex.

The value of Tan is defined in terms of the function Tanh, by $\tan(z) = -i \tanh(iz)$.


z A Complex object.

Returns — A newly constructed Complex initialized to the tangent of the argument.

Tanh

```csharp
```

Returns the hyperbolic tanh of a Complex.

If $z = x + iy$,

- $\tanh(\bar{z}) = -\tanh(z)$ and tanh is odd.
- $\tanh(+0 + i0)$ returns $+0 + i0$.
- $\tanh(+\infty + iy)$ returns $1 + i0$, for all positive-signed numbers $y$.
- $\tanh(x + i\infty)$ returns $NaN + iNaN$, for finite $x$.
- $\tanh(+\infty + iNaN)$ returns $1 \pm i0$ (where the sign of the imaginary part of the result is unspecified).
- $\tanh(NaN + i0)$ returns $NaN + i0$.
- $\tanh(NaN + iy)$ returns $NaN + iNaN$, for all nonzero numbers $y$.
- $\tanh(x + iNaN)$ returns $NaN + iNaN$, for finite $x$.
- $\tanh(NaN + iNaN)$ returns $NaN + iNaN$.

z A Complex object.

Returns — A newly constructed Complex initialized to the hyperbolic tangent of the argument.
Final public string ToString(string format, System.IFormatProvider formatProvider)
Formats the value of the current instance using the specified format.

format    The String specifying the format to use.
formatProvider The IFormatProvider to use to format the value.

ToString
override public string ToString()
Returns a String representation for the specified Complex.

Returns — A String representation for this object.

Description
The binary operations have the form, where op is Add, Subtract, Multiply or Divide.

public static Complex op(Complex x, Complex y) // x op y
public static Complex op(Complex x, double y) // x op y
public static Complex op(double x, Complex y) // x op y

Complex objects are immutable. Once created there is no way to change their value. The functions in this class follow the rules for complex arithmetic as defined C9x Annex G: IEC 559-compatible complex arithmetic. The API is not the same, but handling of infinities, NaNs, and positive and negative zeros is intended to follow the same rules.

Example: Roots of a Quadratic Equation
The two roots of the quadratic equation $ax^2 + bx + c$ are computed using the formula

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

using System;
using Imsl.Math;

public class ComplexEx1
{
    public static void Main(String[] args)
    {
        Complex a = new Complex(2.0, 3.0);
        double b = 4.0;
        Complex c = new Complex(1.0, -2.0);

        Complex disc = Complex.Sqrt(b*b - 4.0*a*c);
        Complex root1 = (-b + disc) / (2.0*a);
        Complex root2 = (-b - disc) / (2.0*a);
Console.Out.WriteLine("Root1 = " + root1);
Console.Out.WriteLine("Root2 = " + root2);
}
}

**Output**

Root1 = 0.195552704020374+0.714335671546131i
Root2 = -0.810937319404989+0.208741251530793i

---

**Physical Structure**

Return the value of various mathematical and physical constants.

`public structure Imsl.Math.Physical`

**Constructors**

`Physical`

`public Physical(double magnitude, string units)`

Constructs a new `Physical` object and initializes this object to a `double` value.

- **magnitude**  A `double` value to which the copy of the object is initialized.
- **units**  A `String` specifying the unit.

`Physical`

`public Physical(double magnitude, int length, int mass, int time, int current, int temperature)`

Constructs a new `Physical` object and initializes this object to a `double` value along with `int` values for length, mass, time, current, and temperature.

- **magnitude**  A `double` value to which this object is initialized.
- **length**  An `int` value assigned to this object’s length.
- **mass**  An `int` value assigned to this object’s mass.
- **time**  An `int` value assigned to this object’s time.
- **current**  An `int` value assigned to this object’s current.
- **temperature**  An `int` value assigned to this object’s temperature.
Operators

+  
static public operator x + y  
Add two compatible Physical objects.  
System.ArgumentException is thrown if x and y are not compatible.

x  A Physical object which is to be added.
y  A Physical object which is to be added.

Returns — A Physical object which is the sum of x + y.

/  
static public operator x / y  
Divide two Physical objects.

x  A Physical object which is the numerator.
y  A Physical object which is the divisor.

Returns — A Physical object which is the result of x/y.

/  
static public operator x / y  
Divide a Physical object by a double.

x  A Physical object which is the numerator.
y  A double object which is the divisor.

Returns — A Physical object which is the result of x/y.

/  
static public operator x / y  
Divide a double by a Physical object.

x  A double which is the numerator.
y  A Physical object which is the divisor.

Returns — A Physical object which is the result of x/y.

==  
static public operator x == y  
Returns true if x and y are equal.

x  A Physical object.
y  A Physical object.

Returns — A boolean value of true if x and y are equal.
static public operator () x
Returns the value of this dimensionless object.

System.ArgumentException is thrown if the this object is not dimensionless

x  A Physical object.

Returns — The double value of the dimensionless object.

!=
static public operator x != y
Returns true if x and y are not equal.

x  A Physical object.
y  A Physical object.

Returns — A boolean value of true if x and y are not equal.

*
static public operator x * y
Multiply two Physical objects.

x  A Physical object which is to be multiplied.
y  A Physical object which is to be multiplied.

Returns — A Physical object which is the product of x and y.

*
static public operator x * y
Multiply a Physical object and a double.

x  A Physical object which is to be multiplied.
y  A double which is to be multiplied.

Returns — A Physical object which is the product of x and y.

*
static public operator x * y
Multiply a double and a Physical object

x  A double which is to be multiplied.
y  A Physical object which is to be multiplied.

Returns — A Physical object which is the product of x and y.

-
static public operator x - y
Subtract two compatible Physical objects.

System.ArgumentException is thrown if x and y are not compatible
- static public operator -\(x\)
  Negate a Physical object.

  x  A Physical object which is to be negated.

  Returns — A Physical object which has been negated.

Methods

Add
  Adds two compatible Physical objects.

  x  A Physical object which is to be added.

  y  A Physical object which is to be added.

  Returns — A Physical object which is the sum of \(x + y\).

  System.ArgumentException is thrown if \(x\) and \(y\) are not compatible

CheckCompatibility
  static public void CheckCompatibility(Imsl.Math.Physical \(x\), Imsl.Math.Physical \(y\))
  Checks the compatibility of two Physical objects.

  x  A Physical object.

  y  A Physical object to be checked against \(x\).

  System.ArgumentException is thrown if the two Physical objects are incompatible

Constant
  static public Imsl.Math.Physical Constant(string \(name\))
  Returns the value of a constant, given its name.

  \(name\)  A String representing the name of the constant to be returned.

  Returns — The Physical object containing the value of the constant, in its default units.

  System.ArgumentException is thrown when the name given is undefined
Constant

static public double Constant(string name, string units)
Returns the value of a constant, given its name, in the specified units.

name  A String representing the name of the constant to be returned.
units  A String representing the units in which the constant is to be returned.

Returns — A double containing the value of the constant in the specified units.

System.ArgumentException is thrown if the constant name is undefined

Convert

Converts a value to a different set of units.

physical  A Physical object specifying the value to be converted.
unitsNew  A String specifying the units to which physical is to be converted.

Returns — A Physical object containing the value of physical converted to the new units.

System.ArgumentException is thrown if the new and old units are incompatible

DefineConstant

static public void DefineConstant(string name, Imsl.Math.Physical magnitude)
Defines a new constant.

name  A String specifying the name of the constant to be defined.
magnitude  A Physical object defining the value of the new constant.

DefinePrefix

static public void DefinePrefix(string name, double magnitude)
Defines a new prefix.

name  A String specifying the name of the prefix to be defined.
magnitude  The double value of the prefix.

DefineUnit

static public void DefineUnit(string name, Imsl.Math.Physical magnitude)
Defines a new unit.

name  A String specifying the name of the unit to be defined.
magnitude  A Physical object defining the value of one unit in terms of SI units.
Divide
    static public Imsl.Math.Physical Divide(Imsl.Math.Physical x,
        Imsl.Math.Physical y)
Divides two Physical objects.
    x     A Physical object which is the numerator.
    y     A Physical object which is the divisor.
Returns — A Physical object which is the result of x/y.

Divide
Divides a Physical object by a double.
    x     A Physical object which is the numerator.
    y     A double object which is the divisor.
Returns — A Physical object which is the result of x/y.

Divide
    static public Imsl.Math.Physical Divide(double x, Imsl.Math.Physical y)
Divides a double by a Physical object.
    x     A double which is the numerator.
    y     A Physical object which is the divisor.
Returns — A Physical object which is the result of x/y.

DoubleValue
    public double DoubleValue()
Returns the value of this dimensionless object.
Returns — The double value of the dimensionless object.

System.ArgumentException is thrown if the this object is not dimensionless

Equals
    override public bool Equals(Object x)
Returns true if x equals this value.
    x     An Object to be tested for equality.
Returns — A boolean value of true if x and this Physical object are equal.

Equals
Returns true if x and y are equal.
    x     A Physical object.
    y     A Physical object.
Returns — A boolean value of true if x and y are equal.

GetHashCode

override public int GetHashCode()
Serves as a hash function for a particular type, suitable for use in hashing algorithms and
data structures like a hash table.
Returns — A hash code for the current Object.

Multiply

Multiply two Physical objects.

x  A Physical object which is to be multiplied.
y  A Physical object which is to be multiplied.
Returns — A Physical object which is the product of x and y.

Multiply

Multiply a Physical object and a double.

x  A Physical object which is to be multiplied.
y  A double which is to be multiplied.
Returns — A Physical object which is the product of x and y.

Multiply

Multiply a double and a Physical object.

x  A double which is to be multiplied.
y  A Physical object which is to be multiplied.
Returns — A Physical object which is the product of x and y.

Negate

Negate a Physical object.

x  A Physical object which is to be negated.
Returns — A Physical object which has been negated.

Subtract

Subtract two compatible Physical objects.

x  A Physical object.
y  A Physical object which is to be subtracted from x.

Returns — A Physical object which is the result of x - y.

System.ArgumentException is thrown if x and y are not compatible

ToString
override public string ToString()
Returns a String containing the value and units, if any.
Returns — A String specifying the value and units, if any, of this Physical object.

UnitsString
public string UnitsString()
Returns a String containing the units only.
Returns — A String specifying the units of this Physical object.

Description

The case of the String specifying the name of the physical constant does not matter. The names 'PI', 'Pi', 'pI' and 'pi' are equivalent.
The units of the physical constants are in SI units, (meter-kilogram-second).
The names allowed are as follows:
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>Atomic mass unit</td>
<td>1.6005402E-27 kg</td>
<td>[1]</td>
</tr>
<tr>
<td>ATM</td>
<td>Standard atm pressure</td>
<td>1.01325E+5 N/m²</td>
<td>E2</td>
</tr>
<tr>
<td>AU</td>
<td>Astronomical unit</td>
<td>1.496E+11 m</td>
<td></td>
</tr>
<tr>
<td>Avogadro</td>
<td>Avogadro's number</td>
<td>6.0221367E+23 /mole</td>
<td></td>
</tr>
<tr>
<td>Boltzman</td>
<td>Boltzman's constant</td>
<td>1.380658E-23 J/K</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Speed of light</td>
<td>2.997924580E+8 m/s</td>
<td>E1</td>
</tr>
<tr>
<td>Catalan</td>
<td>Catalan’s constant</td>
<td>0.915965...</td>
<td>E3</td>
</tr>
<tr>
<td>E</td>
<td>Base of natural logs</td>
<td>2.718...</td>
<td>E3</td>
</tr>
<tr>
<td>ElectronCharge</td>
<td>Electron charge</td>
<td>1.60217733E-19 C</td>
<td></td>
</tr>
<tr>
<td>ElectronMass</td>
<td>Electron mass</td>
<td>9.1093897E-31 kg</td>
<td></td>
</tr>
<tr>
<td>ElectronVolt</td>
<td>Electron volt</td>
<td>1.60217733E-19 J</td>
<td></td>
</tr>
<tr>
<td>Euler</td>
<td>Euler’s constant gamma</td>
<td>0.577...</td>
<td>E3</td>
</tr>
<tr>
<td>Faraday</td>
<td>Faraday constant</td>
<td>9.6485309E+4 C/mole</td>
<td></td>
</tr>
<tr>
<td>FineStructure</td>
<td>Fine structure</td>
<td>7.29735308E-3</td>
<td></td>
</tr>
<tr>
<td>Gamma</td>
<td>Euler’s constant</td>
<td>0.577...</td>
<td>E3</td>
</tr>
<tr>
<td>Gas</td>
<td>Gas constant</td>
<td>8.314510 J/mole/K</td>
<td></td>
</tr>
<tr>
<td>Gravity</td>
<td>Gravitational constant</td>
<td>6.67259E-11 Nm²/kg²</td>
<td></td>
</tr>
<tr>
<td>Hbar</td>
<td>Planck constant / 2 pi</td>
<td>1.05457266E-34 J*s</td>
<td></td>
</tr>
<tr>
<td>PerfectGasVolume</td>
<td>Std vol ideal gas</td>
<td>2.241383E-2 m³/mole</td>
<td></td>
</tr>
<tr>
<td>Pi</td>
<td>Pi</td>
<td>3.141...</td>
<td>E3</td>
</tr>
<tr>
<td>Planck</td>
<td>Planck’s constant h</td>
<td>6.6260755E-34 J*s</td>
<td></td>
</tr>
<tr>
<td>ProtonMass</td>
<td>Proton mass</td>
<td>1.6726231E-27 kg</td>
<td></td>
</tr>
<tr>
<td>Rydberg</td>
<td>Rydberg’s constant</td>
<td>1.0973731534E+7 /m</td>
<td></td>
</tr>
<tr>
<td>SpeedLight</td>
<td>Speed of light</td>
<td>2.997924580E+8 m/s</td>
<td>E1</td>
</tr>
<tr>
<td>StandardGravity</td>
<td>Standard g</td>
<td>9.80665 m/s²</td>
<td>E2</td>
</tr>
<tr>
<td>StandardPressure</td>
<td>Standard atm pressure</td>
<td>1.01325E+5 N/m²</td>
<td>E2</td>
</tr>
<tr>
<td>StefanBoltzmann</td>
<td>Stefan-Boltzman</td>
<td>5.67051E-8 W/K¹/m²</td>
<td></td>
</tr>
<tr>
<td>WaterTriple</td>
<td>Triple point of water</td>
<td>2.7316E+2 K</td>
<td>E2</td>
</tr>
</tbody>
</table>

Units strings have the form U1*U2*...*Um/V1/.../Vn, where Ui and Vi are the names of basic units or are the names of basic units raised to a power. Examples are, 'METER*KILOGRAM/SECOND', 'M*KG/S', 'METER', or 'M/KG²'. These strings are case insensitive.

The basic unit names allowed are as follows.

Units of time
day, hour = hr, min = minute, s = sec = second, year

Units of frequency
Hertz = Hz

Units of mass
AMU, g = gram, lb = pound, ounce = oz, slug

Units of distance
Angstrom, AU, ft = feet = foot, in = inch, m = meter = metre, micron, mile, mill, parsec, yard
Units of area
acre
Units of volume
l = liter = litre
Units of force
dyne, N = Newton, poundal
Units of energy
BTU(thermochemical), Erg, J = Joule
Units of work
W = watt
Units of pressure
ATM = atmsphere, bar, Pascal
Units of temperature
degC = Celsius, degF = Fahrenheit, degK = Kelvin
Units of viscosity
poise, stoke
Units of charge
Abcoulomb, C = Coulomb, statcoulomb
Units of current
A = ampere, abampere, statampere
Units of voltage
Abvolt, V = volt
Units of magnetic induction
T = Tesla, Wb = Weber
Other units
1, farad, mole, Gauss, Henry, Maxwell, Ohm
The following metric prefixes may be used with the above units. Note that the one or two letter prefixes may only be used with one letter unit abbreviations.
A = atto = 1.E-18
F = femto = 1.E-15
P = pico = 1.E-12
N = nano = 1.E-9
U = micro = 1.E-6
M = milli = 1.E-3
C = centi = 1.E-2
D = deci = 1.E-1
DK = deca = 1.E+1
K = kilo = 1.E+3
myria = 1.E+4 (no single letter prefix; M means milli)
mega = 1.E+6 (no single letter prefix; M means milli)
G = giga = 1.E+9
T = tera = 1.E+12

Example: Compute Kinetic Energy

The kinetic energy of a mass in motion is given by

\[ T = \frac{1}{2}mv^2 \]

where \( m \) is the mass and \( v \) is the velocity. In this example the mass is 2.4 pounds and the velocity is 6.7 meters per second. The infix operators defined by \texttt{Physical} automatically handle the unit conversions and computes the current units for the result.

```csharp
using System;
using Imsl.Math;

public class PhysicalEx1
{
    public static void Main(String[] args)
    {
        Physical mass = new Physical(2.4, "pound");
        Physical velocity = new Physical(6.7, "m/s");
        Physical energy = 0.5*mass*velocity*velocity;
        Console.Out.WriteLine("Kinetic energy is " + energy);
    }
}
```

Output

Kinetic energy is 24.43411378716 m\(^2\)/kg/s\(^2\)

Miscellaneous

Physical Structure • 221
EpsilonAlgorithm Class

The class is used to determine the limit of a sequence of approximations, by means of the Epsilon algorithm of P. Wynn.

```csharp
public class Imsl.Math.EpsilonAlgorithm
```

### Property

**ErrorEstimate**

```csharp
public double ErrorEstimate {get; }
```

Returns the current error estimate. A `double` containing the current error estimate.

### Constructors

**EpsilonAlgorithm**

```csharp
public EpsilonAlgorithm()
```

The class is used to determine the limit of a sequence of approximations, by means of the Epsilon algorithm of P. Wynn.

An estimate of the absolute error is also given. The condensed Epsilon table is computed. Only those elements needed for the computation of the next diagonal are preserved.

**EpsilonAlgorithm**

```csharp
public EpsilonAlgorithm(int maxTableSize)
```

The class is used to determine the limit of a sequence of approximations, by means of the Epsilon algorithm of P. Wynn.

An estimate of the absolute error is also given. The condensed Epsilon table is computed. Only those elements needed for the computation of the next diagonal are preserved.

- **maxTableSize** A `int` which specifies the maximum size of Epsilon Table to be computed.

### Method

**Extrapolate**

```csharp
public double Extrapolate(double x)
```

Extrapolates the convergence limit of a sequence.

- **x** A `double` which specifies the next point in the original series.

Returns — A `double` containing the estimate of the limit of the series.
Description

An estimate of the absolute error is also given. The condensed Epsilon table is computed. Only those elements needed for the computation of the next diagonal are preserved.
Chapter 11: Printing Functions

Types

class PrintMatrix ............................................................. 225
class PrintMatrixFormat .................................................... 229
class PrintMatrixFormat.ParsePosition ............................. 232
enumeration PrintMatrix.MatrixType .................................. 232
enumeration PrintMatrixFormat.FormatType ...................... 233
enumeration PrintMatrixFormat.ColumnLabelType ............... 234
enumeration PrintMatrixFormat.RowLabelType .................... 235

PrintMatrix Class

Matrix printing utilities.

public class Imsl.Math.PrintMatrix

Constructors

PrintMatrix
public PrintMatrix()
    Creates an instance of the PrintMatrix class without a title and directs it to the default output stream.
    The matrix is printed without a title to System.Console.Out.

PrintMatrix
public PrintMatrix(System.IO.TextWriter writer)
    Creates an instance of the PrintMatrix class without a title and directs it to a specified output stream.
writer  The TextWriter to which the matrix is to be written.

PrintMatrix
    public PrintMatrix(string title)
    Creates a PrintMatrix object with a title directed to the default output stream.
    The matrix is printed without a title to System.Console.Out.

    title  A String which specifies the title to be printed above the matrix.

PrintMatrix
    public PrintMatrix(System.IO.TextWriter writer, string title)
    Creates a PrintMatrix object with a title directed to a specified output stream.

    writer  A String which specifies the TextWriter to which the matrix is to be written.
    title  The title to be printed above the matrix.

Methods

Print
    void Print(string text)
    Prints a string.
    This function can be overridden to print to something other than a PrintStream.

    text  The String to be printed.

Print
    public void Print(Object array)
    Prints an nRow by nColumn matrix with the default format.

    array  A two-dimensional, non-empty, rectangular Object array.

Print
    public void Print(Imsl.Math.PrintMatrixFormat pmf, Object array)
    Prints an nRow by nColumn matrix with specified format.

    pmf  A PrintMatrixFormat matrix format.
    array  A two-dimensional, non-empty, rectangular Object array.

PrintHTML
    public void PrintHTML(Imsl.Math.PrintMatrixFormat pmf, Object array, int nRows, int nColumns)
    Prints an nRow by nColumn matrix with specified format for HTML output.

    pmf  A PrintMatrixFormat matrix format.
    array  The Matrix to be printed.
    nRows  An int specifying the number of rows in the matrix.
**nColumns**  An int specifying the number of columns in the matrix.

**Println**
```java
void println()
```
Prints a newline.

This function can be overridden to print to something other than a PrintStream.

**SetColumnSpacing**
```java
public Imsl.Math.PrintMatrix SetColumnSpacing(int columnSpacing)
```
Sets the number of spaces between columns.

The default value is 2.

**columnSpacing**  An int specifying the number of spaces between columns.

Returns — The PrintMatrix object.

**SetEqualColumnWidths**
```java
public Imsl.Math.PrintMatrix SetEqualColumnWidths(bool equalColumnWidths)
```
Force all of the columns to have the same width.

**equalColumnWidths**  A boolean which specifies that all column widths will be equal.

Returns — The PrintMatrix object.

**SetMatrixType**
```java
public Imsl.Math.PrintMatrix SetMatrixType(Imsl.Math.PrintMatrix.MatrixType matrixType)
```
Set matrix type.

Values for **matrixType** are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MatrixType.Full</td>
</tr>
<tr>
<td>1</td>
<td>MatrixType.UpperTriangular</td>
</tr>
<tr>
<td>2</td>
<td>MatrixType.LowerTriangular</td>
</tr>
<tr>
<td>3</td>
<td>MatrixType.StrictUpperTriangular</td>
</tr>
<tr>
<td>4</td>
<td>MatrixType.StrictLowerTriangular</td>
</tr>
</tbody>
</table>

**matrixType**  An int specifying the matrix type.

Returns — The PrintMatrix object.

**SetPageWidth**
```java
public Imsl.Math.PrintMatrix SetPageWidth(int pageWidth)
```
Sets the page width.

The default value is the largest possible integer.

**pageWidth**  An int specifying the page width.

Returns — The PrintMatrix object.
setTitle

```csharp
public Imsl.Math.PrintMatrix SetTitle(string title)
```

Sets the matrix title.

title  A String specifying the title of the matrix.

Returns — The PrintMatrix object.

---

**Example: Matrix and PrintMatrix**

The 1 norm of a matrix is found using a method from the Matrix class. The matrix is printed using the PrintMatrix class.

```csharp
using System;
using Imsl.Math;

public class PrintMatrixEx1
{
    public static void Main(String[] args)
    {
        double nrm1;
        double[,] a = {{0.0, 1.0, 2.0, 3.0},
                       {4.0, 5.0, 6.0, 7.0},
                       {8.0, 9.0, 8.0, 1.0},
                       {6.0, 3.0, 4.0, 3.0}};

        // Get the 1 norm of matrix a
        nrm1 = Matrix.OneNorm(a);

        // Construct a PrintMatrix object with a title
        PrintMatrix p = new PrintMatrix("A Simple Matrix");

        // Print the matrix and its 1 norm
        p.Print(a);
        Console.Out.WriteLine("The 1 norm of the matrix is " + nrm1);
    }
}
```

---

**Output**

```
A Simple Matrix
0 1 2 3
0 0 1 2 3
1 4 5 6 7
2 8 9 8 1
3 6 3 4 3

The 1 norm of the matrix is 20
```
PrintMatrixFormat Class

This class can be used to customize the actions of PrintMatrix.

```csharp
public class Imsl.Math.PrintMatrixFormat
```

**Properties**

FirstColumnNumber

```csharp
public int FirstColumnNumber {get; set; }
```

Turns on column labeling with index numbers and sets the index for the label of the first column. The number for the first column label.

This is usually 0 or 1. The default is 0.

FirstRowNumber

```csharp
public int FirstRowNumber {get; set; }
```

Turns on row labeling with index numbers and sets the index for the label of the first row. The number for the first row label.

This is usually 0 or 1. The default is 0.

NumberFormat

```csharp
public string NumberFormat {get; set; }
```

The NumberFormat to be used in formatting double and Complex (p. 193) entries. A String containing the format to be used in Complex entries.

**Constructor**

PrintMatrixFormat

```csharp
public PrintMatrixFormat()
```

Constructs a PrintMatrixFormat object.

**Methods**

Format

```csharp
```

Returns a formatted string.
Note, if type is not FormatType.Entry, pos will be set based on the following criteria. If entry is of type double, the index is the position of the decimal point. If entry is of type int, the index is the position of the end of the formatted integer.

See Also: Imsl.Math.PrintMatrixFormat.FormatType (p. 233)

type The type of string requested. See PrintMatrixFormat.FormatType Enumeration.

entry The entry to be formatted. This is only used if type equals Imsl.Math.PrintMatrixFormat.FormatType.Entry (p. 234). For other values of type, this can be set to null.

row The (0-based) row number of the element to be formatted. This is -1 if there is no row number associated with this request.

col The (0-based) column number of the element to be formatted. This is -1 if there is no column number associated with this request.

pos A ParsePosition object used to indicate the alignment center of the return string. This is used only if type is Imsl.Math.PrintMatrixFormat.FormatType.Entry (p. 234).

Returns — A String to be put into the printed table.

SetColumnLabels
    public void SetColumnLabels(string[] columnLabels)
    Turns on column labeling using the given labels.

    columnLabels An array of Strings to be used as column labels. If there are more columns than labels, the labels are reused.

SetNoColumnLabels
    virtual public void SetNoColumnLabels()
    Turns off column labels.

SetNoRowLabels
    virtual public void SetNoRowLabels()
    Turns off row labels.

Description

By default, entries are formatted using the data type’s ToString method.

See Also

Imsl.Math.PrintMatrix (p. 225)
Example: Matrix Formatting

A simple matrix is printed using the default format with the PrintMatrix class. The PrintMatrixFormat class is then used to change the default format.

```csharp
using System;
using Imsl.Math;

public class PrintMatrixFormatEx1
{
    public static void Main(String[] args)
    {
        double[,] a = {{0.0, 1.0, 2.0, 3.0},
                        {4.0, 5.0, 6.0, 7.0},
                        {8.0, 9.0, 8.0, 1.0},
                        {6.0, 3.0, 4.0, 3.0}};

        // Construct a PrintMatrix object with a title
        PrintMatrix p = new PrintMatrix("A Simple Matrix");

        // Print the matrix
        p.Print(a);

        // Turn row and column labels off
        PrintMatrixFormat mf = new PrintMatrixFormat();
        mf.SetNoRowLabels();
        mf.SetNoColumnLabels();

        // Print the matrix
        p.Print(mf, a);
    }
}
```

Output

```
A Simple Matrix
0 1 2 3
0 0 1 2 3
1 4 5 6 7
2 8 9 8 1
3 6 3 4 3

A Simple Matrix
0 1 2 3
4 5 6 7
8 9 8 1
6 3 4 3
```
PrintMatrixFormat.ParsePosition Class

Tracks the current position during parsing.

```csharp
public class Imsl.Math.PrintMatrixFormat.ParsePosition

Property

Index
```public int Index { get; set; }
Current parse position. An
```int
containing the current index position.

Constructor

ParsePosition
```public ParsePosition(int index)
Creates a ParsePosition.
```index The initial position.

PrintMatrix.MatrixType Enumeration

MatrixType indicates what part of the matrix is to be printed.

```csharp
public enumeration Imsl.Math.PrintMatrix.MatrixType
```

Fields

Full
```public Imsl.Math.PrintMatrix.MatrixType Full
Indicates that the full matrix is to be printed.
```

LowerTriangular
```public Imsl.Math.PrintMatrix.MatrixType LowerTriangular
Indicates that only the lower triangular elements of the matrix are to be printed. The
matrix still must be a rectangular matrix.
```
StrictLowerTriangular
   public Imsl.Math.PrintMatrix.MatrixType StrictLowerTriangular
   Indicates that only the strict lower triangular elements of the matrix are to be printed.
   The matrix still must be a rectangular matrix.

StrictUpperTriangular
   public Imsl.Math.PrintMatrix.MatrixType StrictUpperTriangular
   Indicates that only the strict upper triangular elements of the matrix are to be printed.
   The matrix still must be a rectangular matrix.

UpperTriangular
   public Imsl.Math.PrintMatrix.MatrixType UpperTriangular
   Indicates that only the upper triangular elements of the matrix are to be printed. The
   matrix still must be a rectangular matrix.

---

PrintMatrixFormat.FormatType Enumeration

FormatType specifies the argument to format.

public enumeration Imsl.Math.PrintMatrixFormat.FormatType

Fields

BeginColumnLabel
   public Imsl.Math.PrintMatrixFormat.FormatType BeginColumnLabel
   Indicates that the formatting string for ending a column label is to be returned.

BeginColumnLabels
   public Imsl.Math.PrintMatrixFormat.FormatType BeginColumnLabels
   Indicates that the formatting string for beginning a column label row is to be returned.

BeginEntry
   public Imsl.Math.PrintMatrixFormat.FormatType BeginEntry
   Indicates that the formatted string for beginning an entry is to be returned.

BeginMatrix
   public Imsl.Math.PrintMatrixFormat.FormatType BeginMatrix
   Indicates that the formatting string for beginning a matrix is to be returned.

BeginRow
   public Imsl.Math.PrintMatrixFormat.FormatType BeginRow
   Indicates that the formatting string for beginning a row is to be returned.

BeginRowLabel
   public Imsl.Math.PrintMatrixFormat.FormatType BeginRowLabel
Indicates that the formatting string for beginning a row label is to be returned.

**ColumnLabel**

```
public Imsl.Math.PrintMatrixFormat.FormatType ColumnLabel
```
Indicates that the formatted string for a given column label is to be returned.

**EndColumnLabel**

```
public Imsl.Math.PrintMatrixFormat.FormatType EndColumnLabel
```
Indicates that the formatting string for ending a column label is to be returned.

**EndColumnLabels**

```
public Imsl.Math.PrintMatrixFormat.FormatType EndColumnLabels
```
Indicates that the formatting string for ending a column label row is to be returned.

**EndEntry**

```
public Imsl.Math.PrintMatrixFormat.FormatType EndEntry
```
Indicates that the formatted string for ending an entry is to be returned.

**EndMatrix**

```
public Imsl.Math.PrintMatrixFormat.FormatType EndMatrix
```
Indicates that the formatting string for ending a matrix is to be returned.

**EndRow**

```
public Imsl.Math.PrintMatrixFormat.FormatType EndRow
```
Indicates that the formatting string for ending a row is to be returned.

**EndRowLabel**

```
public Imsl.Math.PrintMatrixFormat.FormatType EndRowLabel
```
Indicates that the formatting string for ending a row label is to be returned.

**Entry**

```
public Imsl.Math.PrintMatrixFormat.FormatType Entry
```
Indicates that the formatted string for a given entry is to be returned.

**RowLabel**

```
public Imsl.Math.PrintMatrixFormat.FormatType RowLabel
```
Indicates that the formatted string for a given row label is to be returned.

---

**PrintMatrixFormat.ColumnHeaderType Enumeration**

Type for column labels.

```
public enumeration Imsl.Math.PrintMatrixFormat.ColumnHeaderType
```

---

234 • PrintMatrixFormat.ColumnHeaderType Enumeration  IMSE C# Numerical Library
Fields

LabelNone
    public Imsl.Math.PrintMatrixFormat.ColumnLabelType LabelNone
    Specifies no column labels will be displayed.

LabelNumber
    public Imsl.Math.PrintMatrixFormat.ColumnLabelType LabelNumber
    Specifies column labels will be an array of ints.

LabelString
    public Imsl.Math.PrintMatrixFormat.ColumnLabelType LabelString
    Specifies column labels will be an array of Strings.

PrintMatrixFormat.RowLabelType Enumeration

Type for row labels.

public enumeration Imsl.Math.PrintMatrixFormat.RowLabelType

Fields

LabelNone
    public Imsl.Math.PrintMatrixFormat.RowLabelType LabelNone
    Specifies no row labels will be displayed.

LabelNumber
    public Imsl.Math.PrintMatrixFormat.RowLabelType LabelNumber
    Specifies row labels will be an array of ints.
Chapter 12: Basic Statistics

Types

- class Summary ................................................................. 238
- class Covariances ............................................................. 246
- enumeration Covariances.MatrixType ..................................... 251
- class NormOneSample ....................................................... 251
- class NormTwoSample ....................................................... 256
- class Sort ................................................................. 265
- class Ranks ............................................................... 272
- enumeration Ranks.Tie ........................................................ 279
- class TableOneWay .......................................................... 279
- class TableTwoWay .......................................................... 284
- class TableMultiWay ......................................................... 288
- class TableMultiWay.TableBalanced ..................................... 294
- class TableMultiWay.TableUnbalanced .................................. 295

Usage Notes

The methods/classes for the computations of basic statistics generally have relatively simple arguments. Most of the methods/classes in this chapter allow for missing values. Missing value codes can be set by using `Double.NaN`.

Several methods/classes in this chapter perform statistical tests. These methods in the classes generally return a "p-value" for the test. The p-value is between 0 and 1 and is the probability of observing data that would yield a test statistic as extreme or more extreme under the assumption of the null hypothesis. Hence, a small p-value is evidence for the rejection of the null hypothesis.
Summary Class

Computes basic univariate statistics.

```csharp
public class Imsl.Stat.Summary
```

**Constructor**

```csharp
Summary
public Summary()
    Constructs a new summary statistics object.
```

**Methods**

GetConfidenceMean
```csharp
public double[] GetConfidenceMean(double p)
    Returns the confidence interval for the mean (assuming normality).
```

- **p**  
  A `double` which specifies the confidence level desired, usually 0.90, 0.95 or 0.99.

  Returns — A `double` array of length 2 which contains the lower and upper confidence limits for the mean.

GetConfidenceVariance
```csharp
public double[] GetConfidenceVariance(double p)
    Returns the confidence interval for the variance (assuming normality).
```

- **p**  
  A `double` which specifies the confidence level desired, usually 0.90, 0.95 or 0.99.

  Returns — A `double` array of length 2 which contains the lower and upper confidence limits for the variance.

GetKurtosis
```csharp
public double GetKurtosis()
    Returns the kurtosis.
```

Returns — A `double` representing the kurtosis.

GetKurtosis
```csharp
static public double GetKurtosis(double[] x)
    Returns the kurtosis of the given data set.
```

- **x**  
  A `double` array containing the data set whose kurtosis is to be found.

  Returns — A `double` which specifies the kurtosis of the given data set.
GetKurtosis
    static public double GetKurtosis(double[] x, double[] weight)
Returns the kurtosis of the given data set and associated weights.

    x  A double array containing the data set whose kurtosis is to be found.
    weight A double array containing the weights associated with the data points x.

Returns — A double which specifies the kurtosis of the given data set.

GetMaximum
    public double GetMaximum()
Returns the maximum.

    Returns — A double representing the maximum.

GetMaximum
    static public double GetMaximum(double[] x)
Returns the maximum of the given data set.

    x  A double array containing the data set whose maximum is to be found.

Returns — A double which specifies the maximum of the given data set.

GetMean
    public double GetMean()
Returns the population mean.

    Returns — A double representing the population mean.

GetMean
    static public double GetMean(double[] x)
Returns the mean of the given data set.

    x  A double array containing the data set whose mean is to be found.

Returns — A double which specifies the mean of the given data set.

GetMean
    static public double GetMean(double[] x, double[] weight)
Returns the mean of the given data set with associated weights.

    x  A double array containing the data set whose mean is to be found.
    weight A double array containing the weights associated with the data points x.

Returns — A double which specifies the mean of the given data set.

GetMedian
    static public double GetMedian(double[] x)
Returns the median of the given data set.

    x  A double array containing the data set whose median is to be found.

Returns — A double which specifies the median of the given data set.
GetMinimum

```csharp
public double GetMinimum()
Returns the minimum.

Returns — A double representing the minimum.
```

GetMinimum

```csharp
static public double GetMinimum(double[] x)
Returns the minimum of the given data set.

x  A double array containing the data set whose minimum is to be found.

Returns — A double which specifies the minimum of the given data set.
```

GetMode

```csharp
static public double GetMode(double[] x)
Returns the mode of the given data set.
Ties are broken at random.

x  A double array containing the data set whose mode is to be found.

Returns — A double which specifies the mode of the given data set.
```

GetSampleStandardDeviation

```csharp
public double GetSampleStandardDeviation()
Returns the sample standard deviation.

Returns — A double representing the sample standard deviation.
```

GetSampleStandardDeviation

```csharp
static public double GetSampleStandardDeviation(double[] x)
Returns the sample standard deviation of the given data set.

x  A double array containing the data set whose sample standard deviation is to be found.

Returns — A double which specifies the sample standard deviation of the given data set.
```

GetSampleStandardDeviation

```csharp
static public double GetSampleStandardDeviation(double[] x, double[] weight)
Returns the sample standard deviation of the given data set and associated weights.

x  A double array containing the data set whose sample standard deviation is to be found.

weight  A double array containing the weights associated with the data points x.

Returns — A double which specifies the sample standard deviation of the given data set.
```
GetSampleVariance

public double GetSampleVariance()
Returns the sample variance.
Returns — A double representing the sample variance.

GetSampleVariance

static public double GetSampleVariance(double[] x)
Returns the sample variance of the given data set.

x  A double array containing the data set whose sample variance is to be found.

Returns — A double which specifies the sample variance of the given data set.

GetSampleVariance

static public double GetSampleVariance(double[] x, double[] weight)
Returns the sample variance of the given data set and associated weights.

x  A double array containing the data set whose sample variance is to be found.
weight  A double array containing the weights associated with the data points x.

Returns — A double which specifies the sample variance of the given data set.

GetSkewness

public double GetSkewness()
Returns the skewness.
Returns — A double representing the skewness.

GetSkewness

static public double GetSkewness(double[] x)
Returns the skewness of the given data set.

x  A double array containing the data set whose skewness is to be found.

Returns — A double which specifies the skewness of the given data set.

GetSkewness

static public double GetSkewness(double[] x, double[] weight)
Returns the skewness of the given data set and associated weights.

x  A double array containing the data set whose skewness is to be found.
weight  A double array containing the weights associated with the data points x.

Returns — A double which specifies the skewness of the given data set.

GetStandardDeviation

public double GetStandardDeviation()
Returns the population standard deviation.
Returns — A double representing the population standard deviation.
GetStandardDeviation
static public double GetStandardDeviation(double[] x)
Returns the population standard deviation of the given data set.

x A double array containing the data set whose standard deviation is to be found.

Returns — A double which specifies the population standard deviation of the given data set.

GetStandardDeviation
static public double GetStandardDeviation(double[] x, double[] weight)
Returns the population standard deviation of the given data set and associated weights.

x A double array containing the data set whose standard deviation is to be found.
weight A double array containing the weights associated with the data points x.

Returns — A double which specifies the population standard deviation of the given data set.

GetVariance
public double GetVariance()
Returns the population variance.

Returns — A double representing the population variance.

GetVariance
static public double GetVariance(double[] x)
Returns the population variance of the given data set.

x A double array containing the data set whose population variance is to be found.

Returns — A double which specifies the population variance of the given data set.

GetVariance
static public double GetVariance(double[] x, double[] weight)
Returns the population variance of the given data set and associated weights.

x A double array containing the data set whose population variance is to be found.
weight A double array containing the weights associated with the data points x.

Returns — A double which specifies the population variance of the given data set.

Update
public void Update(double x)
Adds an observation to the Summary object.

x A double which specifies the data observation to be added.

Update
public void Update(double x, double weight)
Adds an observation and associated weight to the Summary object.
x A `double` which specifies the data observation to be added.

`weight` A `double` which specifies the weight associated with the observation.

Update
```
public void Update(double[] x)
```

Adds a set of observations to the `Summary` object.

- `x` A `double` array of data observations to be added.

Update
```
public void Update(double[] x, double[] weight)
```

Adds a set of observations and associated weights to the `Summary` object.

- `x` A `double` array of data observations to be added.
- `weight` A `double` array of weights associated with the observations.

**Description**

For the data in `x`, `Summary` computes the sample mean, variance, minimum, maximum, and other basic statistics. It also computes confidence intervals for the mean and variance if the sample is assumed to be from a normal population.

Missing values, that is, values equal to NaN (not a number), are excluded from the computations. The sum of the weights is used only in computing the mean (of course, then the weighted mean is used in computing the central moments). The definitions of some of the statistics are given below in terms of a single variable `x`. The `i`-th datum is `x_i`, with corresponding weight `w_i`. If weights are not specified, the `w_i` are identically one. The summation in each case is over the set of valid observations, based on the presence of missing values in the data.

Number of nonmissing observations,

\[ n = \sum f_i \]

Mean,

\[ \bar{x}_w = \frac{\sum f_i w_i x_i}{\sum f_i w_i} \]

Variance,

\[ s^2_w = \frac{\sum f_i w_i (x_i - \bar{x}_w)^2}{n - 1} \]
Skewness,

\[
\frac{\sum f_i w_i (x_i - \bar{x}_w)^3 / n}{\left[ \sum f_i w_i (x_i - \bar{x}_w)^2 / n \right]^{3/2}}
\]

Excess or Kurtosis,

\[
\frac{\sum f_i w_i (x_i - \bar{x}_w)^4 / n}{\left[ \sum f_i w_i (x_i - \bar{x}_w)^2 / n \right]^2} - 3
\]

Minimum,

\[x_{\min} = \min(x_i)\]

Maximum,

\[x_{\max} = \max(x_i)\]

**Example: Summary Statistics**

Summary statistics for a small data set are computed.

```csharp
using System;
using Imsl.Stat;

public class SummaryEx1
{
    internal static readonly double[] data1 =
        new double[]{ 3, 6.4, 2, 1.6, -8, 12,
                      -7, 6.4, 22, 1, 0, -3.2};

    public static void Main(String[] args)
    {
        Summary summary = new Summary();
        summary.Update(data1);

        Console.Out.WriteLine("The minimum is " + summary.GetMinimum());
        Console.Out.WriteLine();
        Console.Out.WriteLine("The maximum is " + summary.GetMaximum());
    }
}
```
The minimum is -8
The maximum is 22
The mean is 3.01666666666667
The variance is 61.7097222222222
The sample variance is 67.319696969697
The standard deviation is 7.85555359107315
The skewness is 0.863222413428583
The kurtosis is 0.567706048385121
The confidence Mean is \{-2.19645146860124, 8.22978480193457\}

The confidence Variance is \{33.7826187272065, 194.068533277244\}

---

**Covariances Class**

Computes the sample variance-covariance or correlation matrix.

```csharp
public class Imsl.Stat.Covariances
```

**Properties**

`MissingValueMethod`

```csharp
public int MissingValueMethod {get; set; }
```

Sets the method used to exclude missing values in x from the computations. An `int` scalar indicating the method to use.

The methods are as follows:

<table>
<thead>
<tr>
<th>MissingValueMethod</th>
<th>Action</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The exclusion is listwise, default. (The entire row of x is excluded if any of the values of the row is equal to the missing value code.)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Raw crossproducts are computed from all valid pairs and means, and variances are computed from all valid data on the individual variables. Corrected crossproducts, covariances, and correlations are computed using these quantities.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Raw crossproducts, means, and variances are computed as in the case of <code>MissingValueMethod = 1</code>. However, corrected crossproducts and covariances are computed only from the valid pairs of data. Correlations are computed using these covariances and the variances from all valid data.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Raw crossproducts, means, variances, and covariances are computed as in the case of <code>MissingValueMethod = 2</code>. Correlations are computed using these covariances, but the variances used are computed from the valid pairs of data.</td>
<td></td>
</tr>
</tbody>
</table>

`Double.NaN` is interpreted as the missing value code.

`NumRowMissing`

```csharp
public int NumRowMissing {get; }
```

Returns the total number of observations that contain any missing values (`Double.NaN`). An `int` scalar containing the total number of observations that contain any missing values (`Double.NaN`).
Observations

```java
public int Observations {get; }
```

Returns the sum of the frequencies. An int scalar containing the sum of the frequencies.

If `MissingValueMethod = 0`, observations with missing values are not included.
Otherwise, all observations are included except for observations with missing values for
the weight or the frequency.

SumOfWeights

```java
public double SumOfWeights {get; }
```

Returns the sum of the weights of all observations. A double scalar containing the sum
of the weights of all observations.

If `MissingValueMethod = 0`, observations with missing values are not included.
Otherwise, all observations are included except for observations with missing values for the
weight or the frequency.

Constructor

Covariances

```java
public Covariances(double[,] x)
```

Constructor for `Covariances`.

- `x` A double matrix containing the data.

`System.ArgumentException` is thrown if `x.GetLength(0)`, and `x.GetLength(1)` are
equal to 0.

Methods

Compute

```java
public double[,] Compute(Imsl.Stat.Covariances.MatrixType matrixType)
```

Computes the matrix.

- `matrixType` A `Covariances.MatrixType` indicating the type of matrix to compute.

Returns — A double matrix containing computed result.

- `Imsl.Stat.TooManyObsDeletedException` is thrown if more observations have been
  deleted than were originally entered
  i.e. the sum of frequencies has become negative

- `Imsl.Stat.MoreObsDelThanEnteredException` is thrown if more observations are being
  deleted from "variance-covariance" matrix than were originally entered.
  The corresponding row,column of the incidence matrix is less than zero.

- `Imsl.Stat.DiffObsDeletedException` is thrown if different observations are being
  deleted than were originally entered.
GetIncidenceMatrix

```java
public int[,] GetIncidenceMatrix()
Returns the incidence matrix.
```

If MissingValueMethod is 0, incidence matrix is 1 x 1 and contains the number of valid observations; otherwise, incidence matrix is `x.GetLength(1) x x.GetLength(1)` and contains the number of pairs of valid observations used in calculating the crossproducts for covariance.

Returns — An int matrix containing the incidence matrix.

GetMeans

```java
public double[] GetMeans()
Returns the means of the variables in x.
The components of the array correspond to the columns of x.
```

Returns — A double array containing the means of the variables in x.

SetFrequencies

```java
public void SetFrequencies(double[] frequencies)
The frequency for each observation.
Default: frequencies[] = 1.
```

`frequencies` A double array of size `x.GetLength(0)` containing the frequency for each observation.

SetWeights

```java
public void SetWeights(double[] weights)
Sets the weight for each observation.
Default: weights[] = 1.
```

`weights` A double array of size `x.GetLength(0)` containing the weight for each observation.

Description

Class Covariances computes estimates of correlations, covariances, or sums of squares and crossproducts for a data matrix x. Weights and frequencies are allowed but not required.

The means, (corrected) sums of squares, and (corrected) sums of crossproducts are computed using the method of provisional means. Let \( x_{ki} \) denote the mean based on \( i \) observations for the \( k \)-th variable, \( f_i \) denote the frequency of the \( i \)-th observation, \( w_i \) denote the weight of the \( i \)-th observations, and \( c_{jki} \) denote the sum of crossproducts (or sum of squares if \( j = k \)) based on \( i \) observations. Then the method of provisional means finds new means and sums of crossproducts as shown in the example below.

The means and crossproducts are initialized as follows:

\[
x_{ko} = 0.0 \quad \text{for } k = 1, \ldots, p
\]
c_{jk0} = 0.0 \text{ for } j, k = 1, \ldots, p

where \( p \) denotes the number of variables. Letting \( x_{k,i+1} \) denote the \( k \)-th variable of observation \( i + 1 \), each new observation leads to the following updates for \( x_{ki} \) and \( c_{jki} \) using the update constant \( r_{i+1} \):

\[
r_{i+1} = \frac{f_{i+1} w_{i+1}}{\sum_{l=1}^{i+1} f_l w_l}
\]

\[
\bar{x}_{k, i+1} = \bar{x}_{ki} + (x_{k, i+1} - \bar{x}_{ki}) r_{i+1}
\]

\[
c_{jki, i+1} = c_{jki} + f_{i+1} w_{i+1} (x_{j, i+1} - \bar{x}_{ji}) (x_{k, i+1} - \bar{x}_{ki}) (1 - r_{i+1})
\]

The default value for weights and frequencies is 1. Means and variances are computed based on the valid data for each variable or, if required, based on all the valid data for each pair of variables.

**Example: Covariances**

This example illustrates the use of Covariances class for the first 50 observations in the Fisher iris data (Fisher 1936). Note that the first variable is constant over the first 50 observations.

```csharp
using System;
using Imsl.Stat;
using PrintMatrix = Imsl.Math.PrintMatrix;
using PrintMatrixFormat = Imsl.Math.PrintMatrixFormat;

public class CovariancesEx1
{
    public static void Main(String[] args)
    {
        double[,] x = {{1.0, 5.1, 3.5, 1.4, .2},
                      {1.0, 4.9, 3.0, 1.4, .2},
                      {1.0, 4.7, 3.2, 1.3, .2},
                      {1.0, 4.6, 3.1, 1.5, .2},
                      {1.0, 5.0, 3.6, 1.4, .2},
                      {1.0, 5.4, 3.9, 1.7, .4},
                      {1.0, 4.6, 3.4, 1.4, .3},
                      {1.0, 5.0, 3.4, 1.5, .2},
                  ...};
```
Covariances co = new Covariances(x);

PrintMatrix pm =
    new PrintMatrix("Sample Variances-covariances Matrix");
PrintMatrixFormat pmf = new PrintMatrixFormat();
    pmf.NumberFormat = "0.0000";
    pmf.SetMatrixType(PrintMatrix.MatrixType.UpperTriangular);

    pm.Print(pmf,
        co.Compute(Covariances.MatrixType.VarianceCovariance));
}
Output

Sample Variances-covariances Matrix

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>1</td>
<td>0.1242</td>
<td>0.0992</td>
<td>0.0164</td>
<td>0.0103</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.1437</td>
<td>0.0117</td>
<td>0.0093</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.0302</td>
<td>0.0061</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0.0111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Covariances.MatrixType Enumeration

Specifies the type of matrix to be computed.

```java
public enumeration Imsl.Stat.Covariances.MatrixType
```

Fields

- **CorrectedSSCP**
  ```java
  public Imsl.Stat.Covariances.MatrixType CorrectedSSCP
  ```
  Indicates corrected sums of squares and crossproducts matrix.

- **Correlation**
  ```java
  public Imsl.Stat.Covariances.MatrixType Correlation
  ```
  Indicates correlation matrix.

- **StdevCorrelation**
  ```java
  public Imsl.Stat.Covariances.MatrixType StdevCorrelation
  ```
  Indicates correlation matrix except for the diagonal elements which are the standard deviations.

- **VarianceCovariance**
  ```java
  public Imsl.Stat.Covariances.MatrixType VarianceCovariance
  ```
  Indicates variance-covariance matrix.

NormOneSample Class

Computes statistics for mean and variance inferences using a sample from a normal population.

```java
public class Imsl.Stat.NormOneSample
```
**Properties**

ChiSquaredTest

```csharp
public double ChiSquaredTest {get; }
```

Returns the test statistic associated with the chi-squared test for variances. A **double** containing the test statistic for the chi-squared test.

The chi-squared test is a test of the hypothesis \( \omega^2 = \omega_0^2 \) where \( \omega_0^2 \) is the null hypothesis value as described in **ChiSquaredTestNull**.

ChiSquaredTestDF

```csharp
public int ChiSquaredTestDF {get; }
```

Returns the degrees of freedom associated with the chi-squared test for variances. A **int** containing the degrees of freedom for the chi-squared test.

The chi-squared test is a test of the hypothesis \( \omega^2 = \omega_0^2 \) where \( \omega_0^2 \) is the null hypothesis value as described in **ChiSquaredTestNull**.

ChiSquaredTestNull

```csharp
public double ChiSquaredTestNull {get; set; }
```

The null hypothesis value for the chi-squared test. A **double** containing the null hypothesis value for the chi-squared test.

The default is 1.0.

ChiSquaredTestP

```csharp
public double ChiSquaredTestP {get; }
```

Returns the probability of a larger chi-squared associated with the chi-squared test for variances. A **double** containing the probability of a larger chi-squared for the chi-squared test for variances.

The chi-squared test is a test of the hypothesis \( \omega^2 = \omega_0^2 \) where \( \omega_0^2 \) is the null hypothesis value as described in **ChiSquaredTestNull**.

ConfidenceMean

```csharp
public double ConfidenceMean {get; set; }
```

The confidence level (in percent) for a two-sided interval estimate of the mean. A **double** containing the confidence level of the mean.

**ConfidenceMean** must be between 0.0 and 1.0 and is often 0.90, 0.95 or 0.99. For a one-sided confidence interval with confidence level c (at least 50 percent), set **ConfidenceMean** = 1.0 - 2.0 * (1.0 - c). If the confidence mean is not specified, a 95-percent confidence interval is computed.

ConfidenceVariance

```csharp
public double ConfidenceVariance {get; set; }
```

The confidence level (in percent) for two-sided interval estimate of the variances. A **double** containing the confidence level of the variance.

**ConfidenceVariance** must be between 0.0 and 1.0 and is often 0.90, 0.95 or 0.99. For a one-sided confidence interval with confidence level c (at least 50 percent), set **ConfidenceVariance** = 1.0 - 2.0 * (1.0 - c). If the confidence mean is not specified, a 95-percent confidence interval is computed.
LowerCIMean

```java
public double LowerCIMean {get; }
Returns the lower confidence limit for the mean. A double containing the lower confidence limit for the mean.
```

LowerCIVariance

```java
public double LowerCIVariance {get; }
Returns the lower confidence limits for the variance. A double containing the lower confidence limits for the variance.
```

Mean

```java
public double Mean {get; }
Returns the mean of the sample. A double containing the mean of x.
```

StdDev

```java
public double StdDev {get; }
Returns the standard deviation of the sample. A double containing the standard deviation of the sample.
```

TTest

```java
public double TTest {get; }
Returns the test statistic associated with the t test. A double containing the test statistic for the t test.
The t test is a test, against a two-sided alternative, of the null hypothesis value described in TTestNull.
```

TTestDF

```java
public int TTestDF {get; }
Returns the degrees of freedom associated with the t test for the mean. A int containing the degrees of freedom for the t test.
The t test is a test, against a two-sided alternative, of the null hypothesis value described in TTestNull.
```

TTestNull

```java
public double TTestNull {get; set; }
Sets the Null hypothesis value for t test for the mean. A double containing the hypothesis value.
TTestNull = 0.0 by default.
```

TTestP

```java
public double TTestP {get; }
Returns the probability associated with the t test of a larger t in absolute value. A double containing the probability for the t test.
The t test is a test, against a two-sided alternative, of the null hypothesis value described in TTestNull.
```

UpperCIMean

```java
public double UpperCIMean {get; }
```
Returns the upper confidence limit for the mean. A `double` containing the upper confidence limit for the mean.

UpperCIVariance

```csharp
public double UpperCIVariance { get; }
```

Returns the upper confidence limits for the variance. A `double` the upper confidence limits for the variance.

**Constructor**

NormOneSample

```csharp
public NormOneSample(double[] x)
```

Constructor to compute statistics for mean and variance inferences using a sample from a normal population.

- `x` A one-dimension `double` array containing the observations.

**Description**

The statistics for mean and variance inferences are computed by using a sample from a normal population, including methods for the confidence intervals and tests for both mean and variance. The definitions of mean and variance are given below. The summation in each case is over the set of valid observations, based on the presence of missing values in the data.

Property `Mean`, returns value

\[
\bar{x} = \frac{\sum x_i}{n}
\]

\[\Delta_d^d Z_t\]

Property `StdDev`, returns value

\[
s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n - 1}}
\]

The property `TTest` returns the t statistic for the two-sided test concerning the population mean which is given by

\[
t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}
\]
where $s$ and $\bar{x}$ are given above. This quantity has a $T$ distribution with $n - 1$ degrees of freedom. The property $TTestDF$ returns the degree of freedom.

Property $ChiSquaredTest$ returns the chi-squared statistic for the two-sided test concerning the population variance which is given by

$$\chi^2 = \frac{(n - 1) s^2}{\sigma_0^2}$$

where $s$ is given above. This quantity has a $\chi^2$ distribution with $n - 1$ degrees of freedom. Property $ChiSquaredTestDF$ returns the degrees of freedom.

**Example 1: NormOneSample**

This example uses data from Devore (1982, p335), which is based on data published in the *Journal of Materials*. There are 15 observations. The hypothesis $H_0 : \mu = 20.0$ is tested. The extremely large $t$ value and the correspondingly small $p$-value provide strong evidence to reject the null hypothesis.

```csharp
using System;
using Imsl.Stat;

public class NormOneSampleEx1
{
    public static void Main(String[] args)
    {
        double mean, stdev, lomean, upmean;
        int df;
        double t, pvalue;
        double[] x = new double[]{ 26.7, 25.8, 24.0, 24.9, 26.4,
                                  25.9, 24.4, 21.7, 24.1, 25.9,
                                  27.3, 26.9, 27.3, 24.8, 23.6};

        /* Perform Analysis*/
        NormOneSample nisamp = new NormOneSample(x);
        mean = nisamp.Mean;
        stdev = nisamp.StdDev;
        lomean = nisamp.LowerCIMean;
        upmean = nisamp.UpperCIMean;
        nisamp.TTestNull = 20.0;
        df = nisamp.TTestDF;
        t = nisamp.TTest;
        pvalue = nisamp.TTestP;

        /* Print results */
        Console.Out.WriteLine("Sample Mean = " + mean);
    }
}
```
Console.Out.WriteLine("Sample Standard Deviation = " + stdev);
Console.Out.WriteLine("95% CI for the mean is " + lomean + " " + upmean);
Console.Out.WriteLine("T Test results");
Console.Out.WriteLine("df = " + df);
Console.Out.WriteLine("t = " + t);
Console.Out.WriteLine("pvalue = " + pvalue);
Console.Out.WriteLine("\n");

/* CI variance */
double ciLoVar = n1samp.LowerCIVariance;
double ciUpVar = n1samp.UpperCIVariance;
Console.Out.WriteLine("CI variance is " + ciLoVar + " " + ciUpVar);

/*chi-squared test */
df = n1samp.ChiSquaredTestDF;
t = n1samp.ChiSquaredTest;
pvalue = n1samp.ChiSquaredTestP;
Console.Out.WriteLine("Chi-squared Test results");
Console.Out.WriteLine("Chi-squared df = " + df);
Console.Out.WriteLine("Chi-squared t = " + t);
Console.Out.WriteLine("Chi-squared pvalue = " + pvalue);

Output
Sample Mean = 25.3133333333333
Sample Standard Deviation = 1.57881812336528
95% CI for the mean is 24.4390129997097 26.187653666957
T Test results
df = 14
t = 13.0340861992294
pvalue = 3.21471738118362E-09

CI variance is 1.33609260499922 6.19986346723949
Chi-squared Test results
Chi-squared df = 14
Chi-squared t = 34.8973333333333
Chi-squared pvalue = 0.0015223176141822

NormTwoSample Class

Computes statistics for mean and variance inferences using samples from two normal populations.

public class Imsl.Stat.NormTwoSample
Properties

ChiSquaredTest
public double ChiSquaredTest {get; }
The test statistic associated with the chi-squared test for common, or pooled, variances. A double containing the test statistic for the chi-squared test.

The chi-squared test is a test of the hypothesis $\omega^2 = \omega_0^2$ where $\omega_0^2$ is the null hypothesis value as described in ChiSquaredTestNull.

ChiSquaredTestDF
public int ChiSquaredTestDF {get; }
The degrees of freedom associated with the chi-squared test for the common, or pooled, variances. A int which specifies the degrees of freedom for the chi-squared test.

The chi-squared test is a test of the hypothesis $\omega^2 = \omega_0^2$ where $\omega_0^2$ is the null hypothesis value as described in ChiSquaredTestNull.

ChiSquaredTestNull
public double ChiSquaredTestNull {get; set; }
The null hypothesis value for the chi-squared test. A double containing the null hypothesis value for the chi-squared test.

The default is 1.0.

ChiSquaredTestP
public double ChiSquaredTestP {get; }
The probability of a larger chi-squared associated with the chi-squared test for common, or pooled, variances. A double containing the probability of a larger chi-squared for the chi-squared test for variances.

The chi-squared test is a test of the hypothesis $\omega^2 = \omega_0^2$ where $\omega_0^2$ is the null hypothesis value as described in ChiSquaredTestNull.

ConfidenceMean
public double ConfidenceMean {get; set; }
The confidence level (in percent) for a two-sided interval estimate of the mean of $x$ - the mean of $y$, in percent. A double containing the confidence level of the mean.

ConfidenceMean must be between 0.0 and 1.0 and is often 0.90, 0.95 or 0.99. For a one-sided confidence interval with confidence level c (at least 50 percent), set ConfidenceMean = $1.0 - 2.0(1.0 - c)$. If the confidence mean is not specified, a 95-percent confidence interval is computed, ConfidenceMean = .95.

ConfidenceVariance
public double ConfidenceVariance {get; set; }
The confidence level (in percent) for two-sided interval estimate of the variances. A double containing the confidence level of the variance.

Under the assumption of equal variances, the pooled variance is used to obtain a two-sided ConfidenceVariance percent confidence interval for the common variance with Imsl.Stat.NormTwoSample.LowerCICommonVariance (p. 258) or
Without making the assumption of equal variances, UnequalVariances (p. 260), the ratio of the variances is of interest. A two-sided ConfidenceVariance percent confidence interval for the ratio of the variance of the first sample to that of the second sample is given by the LowerCIRatioVariance and UpperCIRatioVariance. See UnequalVariances (p. 260) and UpperCIRatioVariance (p. 260). The confidence intervals are symmetric in probability. ConfidenceVariance must be between 0.0 and 1.0 and is often 0.90, 0.95 or 0.99. The default is 0.95.

**DiffMean**

```csharp
public double DiffMean {get; }
```

The difference of means for the two samples. A `double` containing the difference in mean. Value = mean of `x` - mean of `y`

**FTest**

```csharp
public double FTest {get; }
```

The F test value of the F test for equality of variances. A `double` containing the F test value of the F test for equality of variances.

**FTestDFdenominator**

```csharp
public int FTestDFdenominator {get; }
```

The denominator degrees of freedom of the F test for equality of variances. A `int` containing the denominator degrees of freedom.

**FTestDFnumerator**

```csharp
public int FTestDFnumerator {get; }
```

The numerator degrees of freedom of the F test for equality of variances. A `int` containing the numerator degrees of freedom.

**FTestP**

```csharp
public double FTestP {get; }
```

The probability of a larger F in absolute value for the F test for equality of variances, assuming equal variances. A `double` containing the probability of a larger F in absolute value, assuming equal variances.

**LowerCICommonVariance**

```csharp
public double LowerCICommonVariance {get; }
```

The lower confidence limits for the common, or pooled, variance. A `double` containing the lower confidence limits for the variance.

**LowerCIDiff**

```csharp
public double LowerCIDiff {get; }
```

The lower confidence limit for the mean of the first population minus the mean of the second for equal or unequal variances. A `double` containing the lower confidence limit for the mean of the first sample minus the mean of the second sample. If UnequalVariances (p. 260) is `true` then the lower confidence limit for unequal variances will be returned.
LowerCIRatioVariance

public double LowerCIRatioVariance {get; }
The approximate lower confidence limit for the ratio of the variance of the first population
to the second. A double containing the approximate lower confidence limit variance.

MeanX

public double MeanX {get; }
The mean of the first sample, x. A double containing the mean of x.

MeanY

public double MeanY {get; }
The mean of the second sample, y. A double containing the mean of y.

PooledVariance

public double PooledVariance {get; }
The Pooled variance for the two samples. A double containing the Pooled variance for
the two samples.

StdDevX

public double StdDevX {get; }
The standard deviation of the first sample, x. A double containing the standard
deviation of the first sample, x.

StdDevY

public double StdDevY {get; }
The standard deviation of the second sample, y. A double containing the standard
deviation of the second sample, y.

TTest

public double TTest {get; }
The test statistic for the Satterthwaite’s approximation for equal or unequal variances.
A double containing the test statistic for the $t$-test.
If UnequalVariances (p. 260) is true then the test statistic for unequal variances will be
returned.

TTestDF

public double TTestDF {get; }
The degrees of freedom for the Satterthwaite’s approximation for $t$-test for either equal or
unequal variances. A double containing the degrees of freedom for the $t$-test.
If UnequalVariances (p. 260) is true then the degrees of freedom for unequal variances
will be returned.

TTestNull

public double TTestNull {get; set; }
The Null hypothesis value for $t$-test for the mean. A double containing the hypothesis
value. 
TTestNull = 0.0 by default.
TTestP
    public double TTestP {get; }
    The approximate probability of a larger \( t \) for the Satterthwaite's approximation for equal or unequal variances. A double containing the probability for the \( t \)-test.

    If UnequalVariances (p. 260) is true then the approximate probability of a larger \( t \) for unequal variances will be returned.

UnequalVariances
    public bool UnequalVariances {get; set; }
    Specifies whether to return statistics based on equal or unequal variances. A boolean which specifies whether the sample variances are unequal.

    A value of true will cause statistics for unequal variances to be returned. A value of false will cause statistics for equal variances to be returned. The default is to return statistics for equal variances.

UpperCICommonVariance
    public double UpperCICommonVariance {get; }
    The upper confidence limits for the common, or pooled, variance. A double containing the upper confidence limits for the variance.

UpperCIDiff
    public double UpperCIDiff {get; }
    The upper confidence limit for the mean of the first population minus the mean of the second for equal or unequal variances. A double containing the upper confidence limit for the mean of the first sample minus the mean of the second sample.

    If UnequalVariances (p. 260) is true then the upper confidence limit for unequal variances will be returned.

UpperCIRatioVariance
    public double UpperCIRatioVariance {get; }
    The approximate upper confidence limit for the ratio of the variance of the first population to the second. A double containing the approximate upper confidence limit variance.

Constructor

NormTwoSample
    public NormTwoSample(double[] x, double[] y)
    Constructor to compute statistics for mean and variance inferences using samples from two normal populations.

    x  A double array containing the first sample.

    y  A double array containing the second sample.

Methods

DowngradeX
public void DownDateX(double[] x)
Removes the observations in x from the first sample.

x  A double array containing the values to remove from the first sample.

DownDateY
public void DownDateY(double[] y)
Removes the observations in y from the second sample.

y  A double array containing the values to remove from the second sample.

Update
public void Update(double[] x, double[] y)
Concatenates samples x and y to the samples provided in the constructor.

x  A double array containing updates to the first sample.

y  A double array containing updates to the second sample.

UpdateX
public void UpdateX(double[] x)
Concatenates the values in x to the first sample provided in the constructor.

x  A double array containing updates for the first sample.

UpdateY
public void UpdateY(double[] y)
Concatenates the values in y to the second sample provided in the constructor.

y  A double array containing updates for the second sample.

Description

Class NormTwoSample computes statistics for making inferences about the means and variances of two normal populations, using independent samples in x1 and x2. For inferences concerning parameters of a single normal population, see class NormOneSample.

Let \( \mu_1 \) and \( \sigma_1^2 \) be the mean and variance of the first population, and let \( \mu_2 \) and \( \sigma_2^2 \) be the corresponding quantities of the second population. The function contains test confidence intervals for difference in means, equality of variances, and the pooled variance.

The means and variances for the two samples are as follows:

\[
\bar{x}_1 = \left( \sum x_{1i}/n_1 \right), \quad \bar{x}_2 = \left( \sum x_{2i} \right)/n_2
\]

and

\[
s_1^2 = \sum (x_{1i} - \bar{x}_1)^2/(n_1 - 1), \quad s_2^2 = \sum (x_{2i} - \bar{x}_2)^2/(n_2 - 1)
\]
Inferences about the Means

The test that the difference in means equals a certain value, for example, \(\mu_0\), depends on whether or not the variances of the two populations can be considered equal. If the variances are equal and \texttt{meanHypothesis} equals 0, the test is the two-sample \(t\)-test, which is equivalent to an analysis-of-variance test. The pooled variance for the difference-in-means test is as follows:

\[
s^2 = \frac{(n_1 - 1)s_1 + (n_2 - 1)s_2}{n_1 + n_2 - 2}
\]

The \(t\) statistic is as follows:

\[
t = \frac{\bar{x}_1 - \bar{x}_2 - \mu_0}{s\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}
\]

Also, the confidence interval for the difference in means can be obtained by first assigning the unequal variances flag to false. This can be done by setting the \texttt{UnequalVariances} property. The confidence interval can then be obtained by the \texttt{LowerCIDiff} and \texttt{UpperCIDiff} properties.

If the population variances are not equal, the ordinary \(t\) statistic does not have a \(t\) distribution and several approximate tests for the equality of means have been proposed. (See, for example, Anderson and Bancroft 1952, and Kendall and Stuart 1979.) One of the earliest tests devised for this situation is the Fisher-Behrens test, based on Fisher’s concept of fiducial probability. A procedure used in the \texttt{TTest}, \texttt{LowerCIDiff} and \texttt{UpperCIDiff} properties assuming unequal variances are specified is the Satterthwaite’s procedure, as suggested by H.F. Smith and modified by F.E. Satterthwaite (Anderson and Bancroft 1952, p. 83). Set \texttt{UnequalVariances} true to obtain results assuming unequal variances.

The test statistic is

\[
t' = \frac{(\bar{x}_1 - \bar{x}_2 - \mu_0)}{s_d}
\]

where

\[
s_d = \sqrt{\frac{(s_1^2/n_1) + (s_2^2/n_2)}{n_1 - 1 + n_2 - 1}}
\]

Under the null hypothesis of \(\mu_1 - \mu_2 = c\), this quantity has an approximate \(t\) distribution with degrees of freedom \(df\), given by the following equation:

\[
df = \frac{s_d^2}{\frac{(s_1^2/n_1)^2}{n_1 - 1} + \frac{(s_2^2/n_2)^2}{n_2 - 1}}
\]
Inferences about Variances

The F statistic for testing the equality of variances is given by $F = \frac{s_{\text{max}}^2}{s_{\text{min}}^2}$, where $s_{\text{max}}^2$ is the larger of $s_1^2$ and $s_2^2$. If the variances are equal, this quantity has an F distribution with $n_1 - 1$ and $n_2 - 1$ degrees of freedom.

It is generally not recommended that the results of the F test be used to decide whether to use the regular t-test or the modified $t'$ on a single set of data. The modified $t'$ (Satterthwaite’s procedure) is the more conservative approach to use if there is doubt about the equality of the variances.

Example 1: NormTwoSample

This example taken from Conover and Iman (1983, p294), involves scores on arithmetic tests of two grade-school classes.

<table>
<thead>
<tr>
<th>Scores for Standard Group</th>
<th>Scores for Experimental Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>111</td>
</tr>
<tr>
<td>75</td>
<td>118</td>
</tr>
<tr>
<td>77</td>
<td>128</td>
</tr>
<tr>
<td>80</td>
<td>138</td>
</tr>
<tr>
<td>104</td>
<td>140</td>
</tr>
<tr>
<td>110</td>
<td>150</td>
</tr>
<tr>
<td>125</td>
<td>163</td>
</tr>
</tbody>
</table>

The question is whether a group taught by an experimental method has a higher mean score. The difference in means and the t test are output. The variances of the two populations are assumed to be equal. It is seen from the output that there is strong reason to believe that the two means are different ($t$ value of -4.804). Since the lower 97.5-percent confidence limit does not include 0, the null hypothesis is that $\mu_1 \leq \mu_2$ would be rejected at the 0.05 significance level. (The closeness of the values of the sample variances provides some qualitative substantiation of the assumption of equal variances.)
/* Perform Analysis for one sample x2*/
NormTwoSample n2samp = new NormTwoSample(x1, x2);
mean = n2samp.DiffMean;
Console.Out.WriteLine("x1mean-x2mean = " + mean);
Console.Out.WriteLine("X1 mean =" + n2samp.MeanX);
Console.Out.WriteLine("X2 mean =" + n2samp.MeanY);

double pVar = n2samp.PooledVariance;
Console.Out.WriteLine("pooledVar = " + pVar);

double loCI = n2samp.LowerCIDiff;
double upCI = n2samp.UpperCIDiff;
Console.Out.WriteLine("95% CI for the mean is " + loCI + " " + upCI);

loCI = n2samp.LowerCIDiff;
upCI = n2samp.UpperCIDiff;
Console.Out.WriteLine("95% CI for the ueq mean is " + loCI + " " + upCI);

Console.Out.WriteLine("T Test Results");
double tDF = n2samp.TTestDF;
double tT = n2samp.TTest;
double tPval = n2samp.TTestP;
Console.Out.WriteLine("T default = " + tDF);
Console.Out.WriteLine("t = " + tT);
Console.Out.WriteLine("p-value = " + tPval);

double stdevX = n2samp.StdDevX;
double stdevY = n2samp.StdDevY;
Console.Out.WriteLine("stdev x1 =" + stdevX);
Console.Out.WriteLine("stdev x2 =" + stdevY);

} }

Output
x1mean-x2mean = -50.4761904761905
X1 mean =91.8571428571428
X2 mean =142.333333333333
pooledVar = 434.632653061224
95% CI for the mean is -73.0100196252951 -27.9423613270859
95% CI for the ueq mean is -73.0100196252951 -27.9423613270859
T Test Results
T default = 14
t = -4.80436150471634
p-value = 0.000280258365677279
stdev x1 =20.8760514420118
stdev x2 =20.82665965685
Sort Class

A collection of sorting functions.

public class Imsl.Stat.Sort

Constructor

Sort

    public Sort()

Methods

Ascending

    static public void Ascending(double[] ra, int[] iperm)
    Sort an array into ascending order.
    Typically, you would initialize iperm to 0, 1, ...
    
    ra  A double array to be sorted into ascending order.
    iperm  A int array to be sorted using the same permutations applied to ra.

Ascending

    static public void Ascending(int[] ra, int[] iperm)
    Sort an array into ascending order.
    
    ra  an int array to be sorted into ascending order
    iperm  an int array to be sorted using the same permutations applied to ra. Typically, you would initialize this to 0, 1, ...

Ascending

    static public void Ascending(double[] ra)
    Sort an array into ascending order.
    
    ra  A double array to be sorted into ascending order.

Ascending

    static public void Ascending(int[] ra)
    Function to sort an integer array into ascending order.
    
    ra  A int array to be sorted into ascending order.
Ascending

```csharp
static public void Ascending(double[,] ra, int nKeys)
Sort a matrix into ascending order by specified keys.

ra  A double matrix to be sorted into ascending order.

nKeys  A int containing the first nKeys columns of ra to be used as the sorting keys.
```

Ascending

```csharp
static public void Ascending(double[,] ra, int[] indkeys)
Sort a matrix into ascending order by specified keys.

ra  A double matrix to be sorted into ascending order.

indkeys  A int array containing the order the columns of ra are to be sorted.
```

Ascending

```csharp
static public void Ascending(double[,] ra, int nKeys, int[] iperm)
Sort an array into ascending order by specified keys.
Typically, you would initialize iperm to 0, 1, ...

ra  A double array to be sorted into ascending order.

nKeys  A int containing the first nKeys columns of ra to be used as the sorting keys.

iperm  A int array to be sorted using the same permutations applied to ra.
```

Ascending

```csharp
static public void Ascending(double[,] ra, int[] indkeys, int[] iperm)
Sort a matrix into ascending order by specified keys.
Typically, you would initialize iperm to 0, 1, ...

ra  A double matrix to be sorted into ascending order.

indkeys  A int array containing the order the columns of ra are to be sorted.

iperm  A int array to be sorted using the same permutations applied to ra.
```

Descending

```csharp
static public void Descending(double[] ra, int[] iperm)
Sort an array into descending order.
Typically, you would initialize iperm to 0, 1, ...

ra  A double array to be sorted into descending order.

iperm  A int array to be sorted using the same permutations applied to ra.
```

Descending

```csharp
static public void Descending(double[] ra)
Sort an array into descending order.

ra  A double array to be sorted into descending order.
```
Descending

static public void Descending(double[,] ra, int nKeys)
Sorts a matrix into descending order by specified keys.

ra  A double matrix to be sorted into descending order.
nKeys  A int containing the first nKeys columns of ra to be used as the sorting keys.

Descending

static public void Descending(double[,] ra, int[] indkeys)
Sorts a matrix into descending order by specified keys.

ra  A double matrix to be sorted into descending order.
indkeys  A int array containing the order the columns of ra are to be sorted.

Descending

static public void Descending(double[,] ra, int nKeys, int[] iperm)
Sorts an array into descending order by specified keys.

Typically, you would initialize iperm to 0, 1, ...

ra  A double array to be sorted into descending order.
nKeys  A int containing the first nKeys columns of ra to be used as the sorting keys.
iperm  A int array to be sorted using the same permutations applied to ra.

Descending

static public void Descending(double[,] ra, int[] indkeys, int[] iperm)
Sorts a matrix into descending order by specified keys.

Typically, you would initialize iperm to 0, 1, ...

ra  A double matrix to be sorted into descending order.
indkeys  A int array containing the order the columns of ra are to be sorted.
iperm  A int array to be sorted using the same permutations applied to ra.

Description

Class Sort contains ascending and descending methods for sorting elements of an array or a matrix. The array ascending method sorts the elements of an array, A, into ascending order by algebraic value. The array A is divided into two parts by picking a central element T of the array. The first and last elements of A are compared with T and exchanged until the three values appear in the array in ascending order. The elements of the array are rearranged until all elements greater than or equal to the central element appear in the second part of the array and all those less than or equal to the central element appear in the first part. The upper and lower subscripts of one of the segments are saved, and the process continues iteratively on the other segment. When one segment is finally sorted, the process begins again by retrieving the subscripts of another unsorted portion of the array. On completion, \( A_j \leq A_i \) for \( j < i \). For more details, see Singleton (1969), Griffin and Redish (1970), and Petro (1970).
The matrix ascending method sorts the rows of real matrix \( x \) using a particular row in \( x \) as the keys. The sort is algebraic with the first key as the most significant, the second key as the next most significant, etc. When \( x \) is sorted in ascending order, the resulting sorted array is such that the following is true:

- For \( i = 0, 1, \ldots, n \) observations, \( x[i][\text{indices_keys}[0]] \leq x[i+1][\text{indices_keys}[0]] \)
- For \( k = 1, \ldots, n \) keys, \( x[i][\text{indices_keys}[j]] = x[i+1][\text{indices_keys}[j]] \) for \( j = 0, 1, \ldots, k - 1 \), then \( x[i][\text{indices_keys}[k]] = x[i+1][\text{indices_keys}[k]] \)

The observations also can be sorted in descending order.

The rows of \( x \) containing the missing value code NaN in at least one of the specified columns are considered as an additional group. These rows are moved to the end of the sorted \( x \). The sorting algorithm is based on a quicksort method given by Singleton (1969) with modifications by Griffen and Redish (1970) and Petro (1970).

All other methods in this class work off of the ascending methods.

**Example 1: Sorting**

An array is sorted by increasing value. A permutation array is also computed. Note that the permutation array begins at 0 in this example.

```csharp
using System;
using Imsl.Math;
using Imsl.Stat;

public class SortEx1
{
    public static void Main(String[] args)
    {
        double[] ra = new double[] { 10.0, -9.0, 8.0, -7.0, 6.0,
                                    5.0, 4.0, -3.0, -2.0, -1.0};
        int[] iperm = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

        PrintMatrix pm = new PrintMatrix("The Input Array");
        PrintMatrixFormat mf = new PrintMatrixFormat();
        mf.SetNoRowLabels();
        mf.SetNoColumnLabels();
        // Print the array
        pm.Print(mf, ra);
        Console.Out.WriteLine();

        // Sort the array
        Sort.Ascending(ra, iperm);

        pm = new PrintMatrix("The Sorted Array - Lowest to Highest");
        mf = new PrintMatrixFormat();
        mf.SetNoRowLabels();
        mf.SetNoColumnLabels();
```
// Print the array
pm.Print(mf, ra);

pm = new PrintMatrix("The Resulting Permutation Array");
mf = new PrintMatrixFormat();
mf.SetNoRowLabels();
mf.SetNoColumnLabels();
// Print the array
pm.Print(mf, iperm);

Output

The Input Array

10
-9
8
-7
6
5
4
-3
-2
-1

The Sorted Array - Lowest to Highest

-9
-7
-3
-2
-1
4
5
6
8
10

The Resulting Permutation Array

1
3
7
8
9
6
5
4
2
Example 2: Sorting

The rows of a 10 x 3 matrix x are sorted in ascending order using Columns 0 and 1 as the keys. There are two missing values (NaNs) in the keys. The observations containing these values are moved to the end of the sorted array.

```csharp
using System;
using Imsl.Math;
using Imsl.Stat;

class SortEx2
{
    static void Main(string[] args)
    {
        int nKeys = 2;
        double[,] x = {
            {1.0, 1.0, 1.0}, {2.0, 1.0, 2.0},
            {1.0, 1.0, 3.0}, {1.0, 1.0, 4.0},
            {2.0, 2.0, 5.0}, {1.0, 2.0, 6.0},
            {1.0, 2.0, 7.0}, {1.0, 1.0, 8.0},
            {2.0, 2.0, 9.0}, {1.0, 1.0, 9.0}};
        int[] iperm = new int[9] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
        x[4,1] = Double.NaN;
        x[6,0] = Double.NaN;

        PrintMatrix pm = new PrintMatrix("The Input Array");
        PrintMatrixFormat mf = new PrintMatrixFormat();
        mf.SetNoRowLabels();
        mf.SetNoColumnLabels();
        // Print the array
        pm.Print(mf, x);
        Console.Out.WriteLine();
        Sort.Ascending(x, nKeys, iperm);

        pm = new PrintMatrix("The Sorted Array - Lowest to Highest");
        mf = new PrintMatrixFormat();
        mf.SetNoRowLabels();
        mf.SetNoColumnLabels();
        // Print the array
        pm.Print(mf, x);

        pm = new PrintMatrix("The permutation array");
        mf = new PrintMatrixFormat();
        mf.SetNoRowLabels();
        mf.SetNoColumnLabels();
        pm.Print(mf, iperm);
    }
}
```
Output

The Input Array

1 1 1
2 1 2
1 1 3
1 1 4
2 NaN 5
1 2 6
NaN 2 7
1 1 8
2 2 9
1 1 9

The Sorted Array - Lowest to Highest

1 1 1
1 1 9
1 1 3
1 1 4
1 1 8
1 2 6
2 1 2
2 2 9
NaN 2 7
2 NaN 5

The permutation array

0
9
2
3
7
5
1
8
6
4
Ranks Class

Compute the ranks, normal scores, or exponential scores for a vector of observations.

public class Imsl.Stat.Ranks

Properties

Fuzz

    public double Fuzz {get; set; }
    The fuzz factor used in determining ties. A double which represents the fuzz factor.

Random

    public System.Random Random {get; set; }
    The Random object. A Random object used in breaking ties.

TieBreaker

    public Imsl.Stat.Ranks.Tie TieBreaker {get; set; }
    The tie breaker for Ranks. A Tie which represents the tie breaker.

Constructor

Ranks

    public Ranks()
    Constructor for the Ranks class.

Methods

ExpectedNormalOrderStatistic

    static public double ExpectedNormalOrderStatistic(int i, int n)
    Returns the expected value of a normal order statistic.

    i    A int which specifies the rank of the order statistic.
    n    A int which specifies the sample size.

    Returns — A double, the expected value of the i-th order statistic in a sample of size n
    from the standard normal distribution.

GetBlomScores

    public double[] GetBlomScores(double[] x)
    Gets the Blom version of normal scores for each observation.

    x    A double array which contains the observations to be ranked.
Returns — A double array which contains the Blom version of normal scores for each observation in x.

GetNormalScores
   public double[] GetNormalScores(double[] x)
   Gets the expected value of normal order statistics (for tied observations, the average of the expected normal scores).
   For tied observations GetNormalScores returns an average of the expected normal scores.
   
   x   A double array which contains the observations.

   Returns — A double array which contains the expected value of normal order statistics for the observations in x.

GetRanks
   public double[] GetRanks(double[] x)
   Gets the rank for each observation.

   x   A double array which contains the observations to be ranked.

   Returns — A double array which contains the rank for each observation in x.

GetSavageScores
   public double[] GetSavageScores(double[] x)
   Gets the Savage scores. (the expected value of exponential order statistics)

   x   A double array which contains the observations.

   Returns — A double array which contains the Savage scores for the observations in x.
   (the expected value of exponential order statistics)

GetTukeyScores
   public double[] GetTukeyScores(double[] x)
   Gets the Tukey version of normal scores for each observation.

   x   A double array which contains the observations to be ranked.

   Returns — A double array which contains the Tukey version of normal scores for each observation in x.

GetVanDerWaerdenScores
   public double[] GetVanDerWaerdenScores(double[] x)
   Gets the Van der Waerden version of normal scores for each observation.

   x   A double array which contains the observations to be ranked.

   Returns — A double array which contains the Van der Waerden version of normal scores for each observation in x.
The class Ranks can be used to compute the ranks, normal scores, or exponential scores of the data in $X$. Ties in the data can be resolved in four different ways, as specified by property TieBreaker. The type of values returned can vary depending on the member function called:

**GetRanks:** Ordinary Ranks

For this member function, the values output are the ordinary ranks of the data in $X$. If $X[i]$ has the smallest value among those in $X$ and there is no other element in $X$ with this value, then

$$\text{GetRanks}(i) = 1.$$

If both $X[i]$ and $X[j]$ have the same smallest value, then

- if $\text{TieBreaker} = 0$, $\text{Ranks}[i] = \text{GetRanks}[j] = 1.5$;
- if $\text{TieBreaker} = 1$, $\text{Ranks}[i] = \text{Ranks}[j] = 2.0$;
- if $\text{TieBreaker} = 2$, $\text{Ranks}[i] = \text{Ranks}[j] = 1.0$;
- if $\text{TieBreaker} = 3$, $\text{Ranks}[i] = \text{Ranks}[j] = 1.0$.

**GetBlomScores:** Normal Scores, Blom Version

Normal scores are expected values, or approximations to the expected values, of order statistics from a normal distribution. The simplest approximations are obtained by evaluating the inverse cumulative normal distribution function, $\text{Cdf.InverseNormal}$, at the ranks scaled into the open interval $(0, 1)$. In the Blom version (see Blom 1958), the scaling transformation for the rank $r_i$ ($1 \leq r_i \leq n$, where $n$ is the sample size) is

$$\frac{r_i - 3/8}{n + 1/4}.$$

The Blom normal score corresponding to the observation with rank $r_i$ is

$$\Phi^{-1}\left(\frac{r_i - 3/8}{n + 1/4}\right),$$

where $\Phi(\cdot)$ is the normal cumulative distribution function.

Adjustments for ties are made after the normal score transformation. That is, if $X[i]$ equals $X[j]$ (within fuzz) and their value is the $k$-th smallest in the data set, the Blom normal scores are determined for ranks of $k$ and $k + 1$, and then these normal scores are averaged or selected in the manner specified by $\text{TieBreaker}$, which is set by the property TieBreaker. (Whether the transformations are made first or ties are resolved first makes no difference except when averaging is done.)

**GetTukeyScores:** Normal Scores, Tukey Version

In the Tukey version (see Tukey 1962), the scaling transformation for the rank $r_i$ is

$$\frac{r_i - 1/3}{n + 1/3}.$$
Ties are handled in the same way as discussed above for the Blom normal scores.

**GetVanDerWaerdenScores: Normal Scores, Van der Waerden Version**

In the Van der Waerden version (see Lehmann 1975, page 97), the scaling transformation for the rank \( r_i \) is \( r_i / (n + 1) \). The Van der Waerden normal score corresponding to the observation with rank \( r_i \) is

\[
\Phi^{-1} \left( \frac{r_i}{n + 1} \right)
\]

Ties are handled in the same way as discussed above for the Blom normal scores.

**GetNormalScores: Expected Value of Normal Order Statistics**

The member function `GetNormalScores` returns the expected values of the normal order statistics. If the value in \( X[i] \) is the \( k \)-th smallest, then the value output in \( \text{SCORE}[i] \) is \( E(Z_k) \), where \( E(\cdot) \) is the expectation operator and \( Z_k \) is the \( k \)-th order statistic in a sample of size \( x.length \) from a standard normal distribution. Ties are handled in the same way as discussed above for the Blom normal scores.

**GetSavageScores: Savage Scores**

The member function `GetSavageScores` returns the expected values of the exponential order statistics. These values are called Savage scores because of their use in a test discussed by Savage (1956) (see Lehman 1975). If the value in \( X[i] \) is the \( k \)-th smallest, then the \( i \)-th output value output is \( E(Y_k) \), where \( Y_k \) is the \( k \)-th order statistic in a sample of size \( n \) from a standard exponential distribution. The expected value of the \( k \)-th order statistic from an exponential sample of size \( n \) is

\[
\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{n-k+1}
\]

Ties are handled in the same way as discussed above for the Blom normal scores.

**Example: Ranks**

In this data from Hinkley (1977) note that the fourth and sixth observations are tied and that the third and twentieth are tied.

```csharp
using System;
using Imsl.Stat;
using Imsl.Math;

public class RanksEx1
{
    public static void Main(String[] args)
    {
        double[] x = new double[]{0.77, 1.74, 0.81, 1.20, 1.95, 1.20,
```
PrintMatrixFormat mf = new PrintMatrixFormat();
mf.SetNoRowLabels();
mf.SetNoColumnLabels();

Ranks ranks = new Ranks();
double[] score = ranks.GetRanks(x);
new PrintMatrix("The Ranks of the Observations - " +
    "Ties Averaged").Print(mf, score);
Console.Out.WriteLine();
ranks = new Ranks();
score = ranks.GetBlomScores(x);
new PrintMatrix("The Blom Scores of the Observations - " +
    "Highest Score used in Ties").Print(mf, score);
Console.Out.WriteLine();
ranks = new Ranks();
score = ranks.GetTukeyScores(x);
new PrintMatrix("The Tukey Scores of the Observations - " +
    "Lowest Score used in Ties").Print(mf, score);
Console.Out.WriteLine();
ranks = new Ranks();
random.Multiplier = 16807;
ranks.Random = random;
score = ranks.GetVanDerWaerdenScores(x);
new PrintMatrix("The Van Der Waerden Scores of the " +
    "Observations - Ties untied by Random").Print(mf, score);
}

Output

The Ranks of the Observations - Ties Averaged

5
18
6.5
11.5
21
2
15

0.47, 1.43, 3.37, 2.20, 3.00,
3.09, 1.51, 2.10, 0.52, 1.62,
1.31, 0.32, 0.59, 0.81, 2.81,
1.87, 1.18, 1.35, 4.75, 2.48,
0.96, 1.89, 0.90, 2.05;
The Blom Scores of the Observations - Highest Score used in Ties

-1.02410618374162  
0.208663745751154  
-0.775546968322378  
-0.294213138930921  
0.472789120992267  
-0.294213138930921  
-1.60981606718445  
-0.0414437330939966  
1.60981606718445  
0.775546968322378  
1.17581347255003  
1.36087334286719  
0.041437330939965  
0.668002132269574  
-1.36087334286719  
0.124617407947998  
-0.208663745751155  
-2.04028132201041  
-1.17581347255003  
-0.775546968322378  
1.02410618374162  
0.294213138930921  
-0.472789120992267  
-0.124617407947998  
2.04028132201041  
0.892918486444395  
-0.56768639112746  
0.381975767696542  
-0.668002132269574  
0.56768639112746

Basic Statistics  Ranks Class • 277
### The Tukey Scores of the Observations - Lowest Score used in Ties

<table>
<thead>
<tr>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.0200762327862</td>
</tr>
<tr>
<td>0.208082136154993</td>
</tr>
<tr>
<td>-0.89970115508476</td>
</tr>
<tr>
<td>-0.380874057516038</td>
</tr>
<tr>
<td>0.471389465588488</td>
</tr>
<tr>
<td>-0.380874057516038</td>
</tr>
<tr>
<td>-1.59868725959458</td>
</tr>
<tr>
<td>-0.0413298117447387</td>
</tr>
<tr>
<td>1.59868725959458</td>
</tr>
<tr>
<td>0.772935693128221</td>
</tr>
<tr>
<td>1.17060337087942</td>
</tr>
<tr>
<td>1.35372485367826</td>
</tr>
<tr>
<td>0.0413298117447387</td>
</tr>
<tr>
<td>0.665869518001049</td>
</tr>
<tr>
<td>-1.35372485367826</td>
</tr>
<tr>
<td>0.124273282084069</td>
</tr>
<tr>
<td>-0.208082136154993</td>
</tr>
<tr>
<td>-2.01450973381435</td>
</tr>
<tr>
<td>-1.7060337087942</td>
</tr>
<tr>
<td>-0.89970115508476</td>
</tr>
<tr>
<td>1.0200762327862</td>
</tr>
<tr>
<td>0.293381232121193</td>
</tr>
<tr>
<td>-0.471389465588488</td>
</tr>
<tr>
<td>-0.124273282084069</td>
</tr>
<tr>
<td>2.01450973381435</td>
</tr>
<tr>
<td>0.89970115508476</td>
</tr>
<tr>
<td>-0.565948821932863</td>
</tr>
<tr>
<td>0.380874057516038</td>
</tr>
<tr>
<td>-0.665869518001049</td>
</tr>
<tr>
<td>0.665948821932863</td>
</tr>
</tbody>
</table>

### The Van Der Waerden Scores of the Observations - Ties untied by Random

<table>
<thead>
<tr>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.989168627340635</td>
</tr>
<tr>
<td>0.203544231532486</td>
</tr>
<tr>
<td>-0.75272879425817</td>
</tr>
<tr>
<td>-0.372289360465191</td>
</tr>
<tr>
<td>0.460494539103116</td>
</tr>
<tr>
<td>-0.286893916923039</td>
</tr>
<tr>
<td>-1.51792915959428</td>
</tr>
<tr>
<td>-0.0404405085656462</td>
</tr>
<tr>
<td>1.51792915959428</td>
</tr>
<tr>
<td>0.75272879425817</td>
</tr>
<tr>
<td>0.75272879425817</td>
</tr>
<tr>
<td>1.13097760824516</td>
</tr>
<tr>
<td>1.30015343336342</td>
</tr>
<tr>
<td>0.0404405085656462</td>
</tr>
<tr>
<td>0.64932913186466</td>
</tr>
<tr>
<td>-1.30015343336342</td>
</tr>
<tr>
<td>0.121587382750483</td>
</tr>
<tr>
<td>-0.203544231532486</td>
</tr>
<tr>
<td>-1.84859628850141</td>
</tr>
</tbody>
</table>

---

**278 • Ranks Class**

**IMSL C# Numerical Library**
Ranks.Tie Enumeration

Determines how to break a tie.

public enumeration Imsl.Stat.Ranks.Tie

Fields

Average
   public Imsl.Stat.Ranks.Tie Average
   Use the average score in the group of ties.

Highest
   public Imsl.Stat.Ranks.Tie Highest
   Use the highest score in the group of ties.

Lowest
   public Imsl.Stat.Ranks.Tie Lowest
   Use the lowest score in the group of ties.

Random
   public Imsl.Stat.Ranks.Tie Random
   Use one of the group of ties chosen at random.

TableOneWay Class

Tallies observations into a one-way frequency table.
public class Imsl.Stat.TableOneWay

Properties

Maximum
    public double Maximum {get; }
    The maximum value of x. A double containing the maximum data bound.

Minimum
    public double Minimum {get; }
    The minimum value of x. A double containing the minimum data bound.

Constructor

TableOneWay
    public TableOneWay(double[] x, int nIntervals)
    Constructor for TableOneWay.

    x  A double array containing the observations.
    nIntervals  A int scalar containing the number of intervals (bins).

Methods

GetFrequencyTable
    public double[] GetFrequencyTable()
    Returns the one-way frequency table.

    nIntervals  intervals of equal length are used with the initial interval starting with the
    minimum value in x and the last interval ending with the maximum value in x. The
    initial interval is closed on the left and the right. The remaining intervals are open on the
    left and the closed on the right. Each interval is of length (max-min)/nIntervals, where
    max is the maximum value of x and min is the minimum value of x.

    Returns — A double array containing the one-way frequency table.

GetFrequencyTable
    public double[] GetFrequencyTable(double lowerBound, double upperBound)
    Returns a one-way frequency table using known bounds.

    The one-way frequency table is computed using two semi-infinite intervals as the initial
    and last intervals. The initial interval is closed on the right and includes lowerBound as
    its right endpoint. The last interval is open on the left and includes all values greater
    than upperBound. The remaining nIntervals - 2 intervals are each of length
    (upperBound - lowerBound) / (nIntervals - 2) and are open on the left and closed
    on the right. nIntervals must be greater than or equal to 3.

    lowerBound  A double specifies the right endpoint.
upperBound  A double specifies the left endpoint.

Returns — A double array containing the one-way frequency table.

GetFrequencyTableUsingClassmarks
public double[] GetFrequencyTableUsingClassmarks(double[] classmarks)
Returns the one-way frequency table using class marks.

Equally spaced class marks in ascending order must be provided in the array classmarks of length nIntervals. The class marks are the midpoints of each of the nIntervals. Each interval is assumed to have length classmarks[1] - classmarks[0]. nIntervals must be greater than or equal to 2.

classmarks  A double array containing either the cutpoints or the class marks.

Returns — A double array containing the one-way frequency table.

GetFrequencyTableUsingCutpoints
public double[] GetFrequencyTableUsingCutpoints(double[] cutpoints)
Returns the one-way frequency table using cutpoints.

The cutpoints are boundaries that must be provided in the array cutpoints of length nIntervals-1. This option allows unequal interval lengths. The initial interval is closed on the right and includes the initial cutpoint as its right endpoint. The last interval is open on the left and includes all values greater than the last cutpoint. The remaining nIntervals-2 intervals are open on the left and closed on the right. Argument nIntervals must be greater than or equal to 3 for this option.

cutpoints  A double array containing the cutpoints.

Returns — A double array containing the one-way frequency table.

Example: TableOneWay

The data for this example is from Hinkley (1977) and Belleman and Hoaglin (1981). The measurements (in inches) are for precipitation in Minneapolis/St. Paul during the month of March for 30 consecutive years.

The first test uses the default tally method which may be appropriate when the range of data is unknown. The minimum and maximum data bounds are displayed.

The second test computes the table using known bounds, where the lower bound is 0.5 and the upper bound is 4.5. The eight interior intervals each have width (4.5 - 0.5)/(10-2) = 0.5. The 10 intervals are \((-\infty, 0.5], (0.5,1.0],..., (4.0,4.5], (4.5,\infty]\).

In the third test, 10 class marks, 0.25, 0.75, 1.25,...,4.75, are input. This defines the class intervals \((0.0,0.5],[0.5,1.0],..., (4.0,4.5],[4.5,5.0]\). Note that unlike the previous test, the initial and last intervals are the same length as the remaining intervals.

In the fourth test, cutpoints, 0.5,1.0, 1.5, 2.0, ...,4.5, are input to define the same 10 intervals as in the second test. Here again, the initial and last intervals are semi-infinite intervals.
using System;
using Imsl.Stat;

public class TableOneWayEx1
{
    public static void Main(String[] args)
    {
        int nIntervals = 10;

        double[] x = new double[] { 0.77, 1.74, 0.81, 1.20, 1.95,
                                  1.20, 0.47, 1.43, 3.37, 2.20,
                                  3.00, 3.09, 1.51, 2.10, 0.52,
                                  1.62, 1.31, 0.32, 0.59, 0.81,
                                  2.81, 1.87, 1.18, 1.35, 4.75,
                                  2.48, 0.96, 1.89, 0.9, 2.05};
        double[] cutPoints = new double[] { 0.5, 1.0, 1.5, 2.0, 2.5,
                                             3.0, 3.5, 4.0, 4.5};
        double[] classMarks = new double[] { 0.25, 0.75, 1.25, 1.75,
                                             2.25, 2.75, 3.25, 3.75,
                                             4.25, 4.75};

        TableOneWay fTbl = new TableOneWay(x, nIntervals);
        double[] table = fTbl.GetFrequencyTable();
        Console.Out.WriteLine("Example 1 ");
        for (int i = 0; i < table.Length; i++)
            Console.Out.WriteLine(i + "  " + table[i]);
        Console.Out.WriteLine("--------------------------");
        Console.Out.WriteLine("Lower bounds= " + fTbl.Minimum);
        Console.Out.WriteLine("Upper bounds= " + fTbl.Maximum);
        Console.Out.WriteLine("--------------------------");
        /* getFrequencyTable using a set of known bounds */
        table = fTbl.GetFrequencyTable(0.5, 4.5);
        for (int i = 0; i < table.Length; i++)
            Console.Out.WriteLine(i + "  " + table[i]);
        Console.Out.WriteLine("---------------------");
        table = fTbl.GetFrequencyTableUsingClassmarks(classMarks);
        for (int i = 0; i < table.Length; i++)
            Console.Out.WriteLine(i + "  " + table[i]);
        Console.Out.WriteLine("--------------------");
        table = fTbl.GetFrequencyTableUsingCutpoints(cutPoints);
        for (int i = 0; i < table.Length; i++)
            Console.Out.WriteLine(i + "  " + table[i]);
    }
}
Output

Example 1

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

-------------------------------
Lover bounds= 0.32  
Upper bounds= 4.75  
-------------------------------

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

-------------------------------
Lover bounds= 0.32  
Upper bounds= 4.75  
-------------------------------

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

-------------------------------
Lover bounds= 0.32  
Upper bounds= 4.75  
-------------------------------

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
TableTwoWay Class

Tallies observations into a two-way frequency table.

public class Imsl.Stat.TableTwoWay

Properties

MaximumX
   public double MaximumX {get; }
   The maximum value of x. A double containing the maximum data bound for x.

MaximumY
   public double MaximumY {get; }
   The maximum value of y. A double containing the maximum data bound for y.

MinimumX
   public double MinimumX {get; }
   The minimum value of x. A double containing the minimum data bound for x.

MinimumY
   public double MinimumY {get; }
   The minimum value of y. A double containing the minimum data bound for y.

Constructor

TableTwoWay
   public TableTwoWay(double[] x, int xIntervals, double[] y, int yIntervals)
   Constructor for TableTwoWay.

   x   A double array containing the data for the first variable.
   xIntervals   A int scalar containing the number of intervals (bins) for variable x.
   y   A double array containing the data for the second variable.
   yIntervals   A int scalar containing the number of intervals (bins) for variable y.

Methods

GetFrequencyTable
   public double[,] GetFrequencyTable()
   Returns the two-way frequency table.

   Intervals of equal length are used. Let xmin and xmax be the minimum and maximum
   values in x, respectively, with similar meanings for ymin and ymax. Then, the first row of
the output table is the tally of observations with the x value less than or equal to $x_{\text{min}} + (x_{\text{max}} - x_{\text{min}})/x_{\text{Intervals}}$, and the y value less than or equal to $y_{\text{min}} + (y_{\text{max}} - y_{\text{min}})/y_{\text{Intervals}}$.

Returns — A two-dimensional double array containing the two-way frequency table.

GetFrequencyTable

public double[,] GetFrequencyTable(double xLowerBound, double xUpperBound, double yLowerBound, double yUpperBound)

Compute a two-way frequency table using intervals of equal length and user supplied upper and lower bounds, xLowerBound, xUpperBound, yLowerBound, yUpperBound. The first and last intervals for both variables are semi-infinite in length. xIntervals and yIntervals must be greater than or equal to 3.

xLowerBound A double specifies the right endpoint for x.

xUpperBound A double specifies the left endpoint for x.

yLowerBound A double specifies the right endpoint for y.

yUpperBound A double specifies the left endpoint for y.

Returns — A two dimensional double array containing the two-way frequency table.

GetFrequencyTableUsingClassmarks

public double[,] GetFrequencyTableUsingClassmarks(double[] cx, double[] cy)

Returns the two-way frequency table using either cutpoints or class marks.

Cutpoints are boundaries and class marks are the midpoints of xIntervals and yIntervals.

Equally spaced class marks in ascending order must be provided in the arrays cx and cy. The class marks the midpoints of each interval. Each interval is taken to have length $cx[1] - cx[0]$ in the x direction and $cy[1] - cy[0]$ in the y direction. The total number of elements in the output table may be less than the number of observations of input data. Arguments xIntervals and yIntervals must be greater than or equal to 2 for this option.

cx A double array containing either the cutpoints or the class marks for x.

cy A double array containing either the cutpoints or the class marks for y.

Returns — A two dimensional double array containing the two-way frequency table.

GetFrequencyTableUsingCutpoints

public double[,] GetFrequencyTableUsingCutpoints(double[] cx, double[] cy)

Returns the two-way frequency table using cutpoints.

The cutpoints (boundaries) must be provided in the arrays cx and cy, of length $(x_{\text{Intervals}}-1)$ and $(y_{\text{Intervals}}-1)$ respectively. The first row of the output table is the tally of observations for which the x value is less than or equal to cx[0], and the y value is less than or equal to cy[0]. This option allows unequal interval lengths. Arguments cx and cy must be greater than or equal to 2.

cx A double array containing either the cutpoints or the class marks for x.
A double array containing either the cutpoints or the class marks for y.

Returns — A two dimensional double array containing the two-way frequency table.

Example: TableTwoWay

The data for x in this example is from Hinkley (1977) and Belleman and Hoaglin (1981). The measurements (in inches) are for precipitation in Minneapolis/St. Paul during the month of March for 30 consecutive years. The data for y were created by adding small integers to the data in x.

The first test uses the default tally method which may be appropriate when the range of data is unknown. The minimum and maximum data bounds are displayed.

The second test computes the table using known bounds, where the x lower, x upper, y lower, y upper bounds are chosen so that the intervals will be 0 to 1, 1 to 2, and so on for x and 1 to 2, 2 to 3 and so on for y.

In the third test, the class boundaries are input at the same intervals as in the second test. The first element of cmx and cmy specify the first cutpoint between classes.

The fourth test uses the cutpoints tally option with cutpoints such that the intervals are specified as in the previous tests.

```csharp
using System;
using Imsl.Stat;
using PrintMatrix = Imsl.Math.PrintMatrix;

public class TableTwoWayEx1
{
    public static void Main(String[] args)
    {
        int nx = 5;
        int ny = 6;

        double[] x = new double[]{ 0.77, 1.74, 0.81, 1.20, 1.95,
                                   1.20, 0.47, 1.43, 3.37, 2.20,
                                   3.00, 3.09, 1.51, 2.10, 0.52,
                                   1.62, 1.31, 0.32, 0.59, 0.81,
                                   2.81, 1.87, 1.18, 1.35, 4.75,
                                   2.48, 0.96, 1.89, 0.9, 2.05};

        double[] y = new double[]{ 1.77, 3.74, 3.81, 2.20, 3.95,
                                   4.20, 1.47, 3.43, 6.37, 3.20,
                                   5.00, 6.09, 2.51, 4.10, 3.52,
                                   2.62, 3.31, 3.32, 1.59, 2.81,
                                   5.81, 2.87, 3.18, 4.35, 5.75,
                                   4.48, 3.96, 2.89, 2.9, 5.05};

        TableTwoWay fTbl = new TableTwoWay(x, nx, y, ny);
        double[,] table = fTbl.GetFrequencyTable();
        Console.Out.WriteLine("Example 1 ");
```
Example 1
Use Min and Max for bounds

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

--------------------------

Lower xbounds= 0.32
Upper xbounds= 4.75
Lower ybounds= 1.47
Upper ybounds= 6.37
--------------------------

Output

Basic Statistics

TableTwoWay Class • 287
TableMultiWay Class

Tallies observations into a multi-way frequency table.

public class Imsl.Stat.TableMultiWay

Properties

BalancedTable

public Imsl.Stat.TableMultiWay.TableBalanced BalancedTable {get; }
An object containing the balanced table. A TableMultiWay.TableBalanced object.

UnbalancedTable

public Imsl.Stat.TableMultiWay.TableUnbalanced UnbalancedTable {get; }
An object containing the unbalanced table. A TableMultiWay.TableUnBalanced object.
Constructors

TableMultiWay
public TableMultiWay(double[,] x, int nKeys)
Constructor for TableMultiWay.

x  A double matrix containing the observations and variables.
nKeys  A int array containing the variables(columns) for which computations are to be performed.

TableMultiWay
public TableMultiWay(double[,] x, int[] indkeys)
Constructor for TableMultiWay.

x  A double matrix containing the observations and variables.
indkeys  A int array containing the variables(columns) for which computations are to be performed.

Methods

GetGroups
public int[] GetGroups()
Returns the number of observations (rows) in each group.

The number of groups is the length of the returned array. A group contains observations in x that are equal with respect to the method of comparison. If n contains the returned integer array, then the first n[0] rows of the sorted x are group number 1. The next n[1] rows of the sorted x are group number 2, etc. The last n[n.length - 1] rows of the sorted x are group number n.length.

Returns — A int array containing the number of observations (row) in each group.

SetFrequencies
public void SetFrequencies(double[] frequencies)
Sets the frequencies for each observation in x.
Length of input must be the same as the number of observations or number of rows in x.
Default frequencies[] = 1.

frequencies  A double array containing the frequency for each observation in x.

Description

The TableMultiWay class determines the distinct values in multivariate data and computes frequencies for the data. This class accepts the data in the matrix x, but performs computations only for the variables (columns) in the first nkeys columns of x or by the variables specified in indkeys. In general, the variables for which frequencies should be
computed are discrete; they should take on a relatively small number of different values. Variables that are continuous can be grouped first. `TableMultiWay` can be used to group variables and determine the frequencies of groups.

The read-only property `BalancedTable` returns a `TableBalanced` object. Its `GetValues` method returns an array with the unique values in the vector of the variables and tallies the number of unique values of each variable table. Each combination of one value from each variable forms a cell in a multi-way table. The frequencies of these cells are entered in a table so that the first variable cycles through its values exactly once, and the last variable cycles through its values most rapidly. Some cells cannot correspond to any observations in the data; in other words, "missing cells" are included in table and have a value of 0.

The read-only property `UnbalancedTable` returns a `TableUnbalanced` object. The frequency of each cell is entered in the unbalanced table so that the first variable cycles through its values exactly once and the last variable cycles through its values most rapidly. `table` is returned by `UnbalancedTable` property. All cells have a frequency of at least 1, i.e., there is no "missing cell." The array `listCells`, returned by method `GetListCells` can be considered "parallel" to `table` because row i of `listCells` is the set of `nkeys` values that describes the cell for which row i of `table` contains the corresponding frequency.

**Example 1: TableMultiWay**

The same data used in SortEx2 is used in this example. It is a 10 x 3 matrix using Columns 0 and 1 as keys. There are two missing values (NaNs) in the keys. NaN is displayed as a ?. Table MultiWay determines the number of groups of different observations.

```csharp
using System;
using Imsl.Stat;
using Imsl.Math;

public class TableMultiWayEx1
{
    public static void Main(String[] args)
    {
        int nKeys = 2;
        double[,] x = {
            {1.0, 1.0, 1.0}, {2.0, 1.0, 2.0},
            {1.0, 1.0, 3.0}, {1.0, 1.0, 4.0},
            {2.0, 2.0, 5.0}, {1.0, 2.0, 6.0},
            {1.0, 2.0, 7.0}, {1.0, 1.0, 8.0},
            {2.0, 2.0, 9.0}, {1.0, 1.0, 9.0}};

        x[4,1] = Double.NaN;
        x[6,0] = Double.NaN;

        PrintMatrix pm = new PrintMatrix("The Input Array");
        PrintMatrixFormat mf = new PrintMatrixFormat();
        mf.SetNoRowLabels();
        mf.SetNoColumnLabels();
        // Print the array
        pm.Print(mf, x);
    }
}
```
```csharp
Console.Out.WriteLine();

TableMultiWay tbl = new TableMultiWay(x, nKeys);
int[] ngroups = tbl.GetGroups();
Console.Out.WriteLine("ngroups");
for (int i = 0; i < ngroups.Length; i++)
    Console.Out.Write(ngroups[i] + " ");
}

Output

The Input Array

1 1 1
2 1 2
1 1 3
1 1 4
2 NaN 5
1 2 6
NaN 2 7
1 1 8
2 2 9
1 1 9

ngroups
5 1 1 1

Example 2: TableMultiWay

The table of frequencies for a data matrix of size 30 x 2 is output.

using System;
using Imsl.Stat;
using Imsl.Math;

public class TableMultiWayEx2
{
    public static void Main(String[] args)
    {
        int[] indkeys = new int[]{0, 1};
        double[,] x = {
            {0.5, 1.5}, {1.5, 3.5},
            {0.5, 3.5}, {1.5, 2.5},
            {1.5, 3.5}, {1.5, 4.5},
            {0.5, 1.5}, {1.5, 3.5},
            {3.5, 6.5}, {2.5, 3.5},
            {2.5, 4.5}, {3.5, 6.5},
            {1.5, 2.5}, {2.5, 4.5},
            {0.5, 3.5}, {1.5, 2.5},
```
TableMultiWay tbl = new TableMultiWay(x, indkeys);

int[] nvalues = tbl.BalancedTable.GetNvalues();
double[] values = tbl.BalancedTable.GetValues();

Console.Out.WriteLine(" row values");
for (int i = 0; i < nvalues[0]; i++)
    Console.Out.Write(values[i] + " ");
Console.Out.WriteLine();

Console.Out.WriteLine(" column values");
for (int i = 0; i < nvalues[1]; i++)
    Console.Out.Write(values[i + nvalues[0]] + " ");
double[] table = tbl.BalancedTable.GetTable();

Console.Out.WriteLine(" Table");

for (int i = 0; i < nvalues[1]; i++)
    Console.Out.Write(values[i] + " ");
for (int j = 0; j < nvalues[1]; j++)
    Console.Out.Write(table[j + (nvalues[1] * i)] + " ");
Console.Out.WriteLine;

Output

row values
0.5 1.5 2.5 3.5 4.5

column values
1.5 2.5 3.5 4.5 5.5 6.5
Example 3: TableMultiWay

The unbalanced table of frequencies for a data matrix of size 4 x 3 is output.

```csharp
using System;
using Imsl.Stat;
using Imsl.Math;

public class TableMultiWayEx3
{
    public static void Main(String[] args)
    {
        int[] indkeys = new int[]{0, 1};
        double[,] x = {
            {2.0, 5.0, 1.0},
            {1.0, 5.0, 2.0},
            {1.0, 6.0, 3.0},
            {2.0, 6.0, 4.0}};
        double[] frq = new double[]{1.0, 2.0, 3.0, 4.0};
        TableMultiWay tbl = new TableMultiWay(x, indkeys);
        tbl.SetFrequencies(frq);

        int ncells = tbl.UnbalancedTable.NCells;
        double[] listCells = tbl.UnbalancedTable.GetListCells();
        double[] table = tbl.UnbalancedTable.GetTable();

        PrintMatrix pm = new PrintMatrix("List Cells");
        PrintMatrixFormat mf = new PrintMatrixFormat();
        mf.SetNoRowLabels();
        mf.SetNoColumnLabels();
        // Print the array
        pm.Print(mf, listCells);
        Console.Out.WriteLine();

        pm = new PrintMatrix("Unbalanced Table");
        mf = new PrintMatrixFormat();
        mf.SetNoRowLabels();
        mf.SetNoColumnLabels();
        // Print the array
        pm.Print(mf, table);
        Console.Out.WriteLine();
    }
}
```
Output

List Cells

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Unbalanced Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

TableMultiWay.TableBalanced Class

Tallies the number of unique values of each variable.

public class Imsl.Stat.TableMultiWay.TableBalanced

Methods

GetNvalues

public int[] GetNvalues()

Returns an array of length nkeys containing in its i-th element (i=0,1,...,nkeys-1), the number of levels or categories of the i-th classification variable (column).

Returns — A int array containing the number of levels or for each variable (column) in x.

GetTable

public double[] GetTable()

Returns an array containing the frequencies for each variable.

The array is of length nValues[0] x nValues[1] x ... x nValues[nkeys] containing the frequencies in the cells of the table to be fit, where nValues contains the result from getNValues.
Empty cells are included in table, and each element of table is nonnegative. The cells of
table are sequenced so that the first variable cycles through its \( nValues[0] \) categories one
time, the second variable cycles through its \( nValues[1] \) categories \( nValues[0] \) times, the
third variable cycles through its \( nValues[2] \) categories \( nValues[0] \times nValues[1] \) times, etc., up to the
\( nkeys \)-th variable, which cycles through its \( nValues[nkeys - 1] \) categories \( nValues[0] \times nValues[1] \times \ldots \times nValues[nkeys - 2] \) times.

Returns — A double array containing the frequencies for each variable in \( x \).

GetValues

\[
\text{public double[]} \text{ GetValues()}
\]
Returns the values of the classification variables.

GetValues returns an array of length \( nValues[0] + nValues[1] + \ldots + nValues[nkeys - 1] \). The first \( nValues[0] \) elements contain the values for the first
classification variable. The next \( nValues[1] \) contain the values for the second variable.
The last \( nValues[nkeys - 1] \) positions contain the values for the last classification
variable, where \( nValues \) contains the result from getNValues.

Returns — A double array containing the values of the classification variables.

---

**TableMultiWay.TableUnbalanced Class**

Tallies the frequency of each cell in \( x \).

public class Imsl.Stat.TableMultiWay.TableUnbalanced

**Property**

NCells

\[
\text{public int NCells \{get; \}}
\]
Returns the number of non-empty cells. A int containing the number of non-empty cells.

**Methods**

GetListCells

\[
\text{public double[]} \text{ GetListCells()}
\]
Returns for each row, a list of the levels of \( nkeys \) coorresponding classification variables
that describe a cell.

Returns — A double array containing the list of levels of \( nkeys \) corresponding
classification variables that describe a cell.

GetTable

\[
\text{public double[]} \text{ GetTable()}
\]
Returns the frequency for each cell.

Returns — A double array containing the frequency for each cell.
Chapter 13: Regression

Types

class LinearRegression ................................................................. 297
class LinearRegression.CoefficientTTestsValue .................................. 301
class NonlinearRegression ............................................................ 302
interface NonlinearRegression.IDerivative ....................................... 314
interface NonlinearRegression.IFunction ....................................... 315
class SelectionRegression ............................................................ 315
enumeration SelectionRegression.Criterion ..................................... 326
class SelectionRegression.SummaryStatistics .................................... 326
class StepwiseRegression ............................................................. 327
class StepwiseRegression.CoefficientTTestsValue ................................ 335
interface StepwiseRegression.Direction ......................................... 336
class UserBasisRegression ............................................................ 337
interface IRegressionBasis ........................................................... 339

LinearRegression Class

Fits a multiple linear regression model with or without an intercept.

public class Imsl.Stat.LinearRegression

Properties

ANOVA
    public Imsl.Stat.ANOVA ANOVA {get; }

CoefficientTTests

```csharp
public Imsl.Stat.LinearRegression.CoefficientTTestsValue CoefficientTTests {
    get;
}
```

Returns statistics relating to the regression coefficients. A
`LinearRegression.CoefficientTTestsValue` object which possesses regression statistics related to the coefficients.

HasIntercept

```csharp
public bool HasIntercept {
    get;
}
```

A `bool` which indicates whether or not an intercept is in this regression model.

Rank

```csharp
public int Rank {
    get;
}
```

Returns the rank of the matrix. An `int` which specifies the matrix rank.

**Constructor**

```csharp
LinearRegression

public LinearRegression(int nVariables, bool hasIntercept)

Constructs a new linear regression object.
```

- `nVariables` An `int` which specifies the number of regression variables.
- `hasIntercept` A `bool` which indicates whether or not an intercept is in this regression model.

**Methods**

GetCoefficients

```csharp
public double[] GetCoefficients()

Returns the regression coefficients.
```

If `HasIntercept` is `false` its length is equal to the number of variables. If `HasIntercept` is `true` then its length is the number of variables plus one and the 0-th entry is the value of the intercept.

Returns — A `double` array containing the regression coefficients.

`Imsl.Math.SingularMatrixException` is thrown when the regression matrix is singular

GetR

```csharp
public double[,] GetR()

Returns a copy of the $R$ matrix.
```

$R$ is the upper triangular matrix containing the $R$ matrix from a QR decomposition of the matrix of regressors.

Returns — A `double` matrix containing a copy of the $R$ matrix.
Update
public void Update(double[] x, double y)
Updates the regression object with a new observation.
x.length must be equal to the number of variables set in the constructor.
x A double array containing the independent (explanatory) variables.
y A double representing the dependent (response) variable.

Update
public void Update(double[] x, double y, double w)
Updates the regression object with a new observation and weight.
x.length must be equal to the number of variables set in the constructor.
x A double array containing the independent (explanatory) variables.
y A double representing the dependent (response) variable.
w A double representing the weight.

Update
public void Update(double[,] x, double[] y)
Updates the regression object with a new set of observations.
The number of rows in x must equal y.length and the number of columns must be equal
to the number of variables set in the constructor.
x A double matrix containing the independent (explanatory) variables.
y A double array containing the dependent (response) variables.

Update
public void Update(double[,] x, double[] y, double[] w)
Updates the regression object with a new set of observations and weights.
The number of rows in x must equal y.length and the number of columns must be equal
to the number of variables set in the constructor.
x A double matrix containing the independent (explanatory) variables.
y A double array containing the dependent (response) variables.
w A double array representing the weights.

Description
If the constructor argument hasIntercept is true, the multiple linear regression model is

\[ y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_k x_{ik} + \varepsilon_i \quad i = 1, 2, \ldots, n \]

where the observed values of the \( y_i \)'s constitute the responses or values of the dependent
variable, the \( x_{i1} \)'s, \( x_{i2} \)'s, \ldots, \( x_{ik} \)'s are the settings of the independent variables, \( \beta_0, \beta_1, \ldots, \beta_k \)
are the regression coefficients, and the \( e_i \)'s are independently distributed normal errors each with mean zero and variance \( \sigma^2 \). If \texttt{hasIntercept} is false, \( \beta_0 \) is not included in the model.

\texttt{LinearRegression} computes estimates of the regression coefficients by minimizing the sum of squares of the deviations of the observed response \( y_i \) from the fitted response

\[
\hat{y}_i
\]

for the observations. This minimum sum of squares (the error sum of squares) is in the ANOVA output and denoted by

\[
SSE = \sum_{i=1}^{n} w_i (y_i - \hat{y}_i)^2
\]

In addition, the total sum of squares is output in the ANOVA table. For the case, \texttt{HasIntercept} is true; the total sum of squares is the sum of squares of the deviations of \( y_i \) from its mean

\[
\bar{y}
\]

—the so-called \textit{corrected total sum of squares}; it is denoted by

\[
SST = \sum_{i=1}^{n} w_i (y_i - \bar{y})^2
\]

For the case \texttt{HasIntercept} is false, the total sum of squares is the sum of squares of \( y_i \)–the so-called \textit{uncorrected total sum of squares}; it is denoted by

\[
SST = \sum_{i=1}^{n} y_i^2
\]

See Draper and Smith (1981) for a good general treatment of the multiple linear regression model, its analysis, and many examples.

In order to compute a least-squares solution, \texttt{LinearRegression} performs an orthogonal reduction of the matrix of regressors to upper triangular form. Givens rotations are used to reduce the matrix. This method has the advantage that the loss of accuracy resulting from forming the crossproduct matrix used in the normal equations is avoided, while not requiring the storage of the full matrix of regressors. The method is described by Lawson and Hanson, pages 207-212.

**Example: Linear Regression**

The coefficients of a simple linear regression model, without an intercept, are computed.
using System;
using Imsl.Stat;

public class LinearRegressionEx1
{
    public static void Main(String[] args)
    {
        // y = 4*x0 + 3*x1
        LinearRegression r = new LinearRegression(2, false);
        double[] c = new double[]{4, 3};
        double[] x0 = {1, 5};
        double[] x1 = {0, 2};
        double[] x2 = {-1, 4};

        r.Update(x0, 1 * c[0] + 5 * c[1]);
        r.Update(x1, 0 * c[0] + 2 * c[1]);
        r.Update(x2, -1 * c[0] + 4 * c[1]);
        double[] coef = r.GetCoefficients();
        Console.Out.WriteLine
            ("The computed regression coefficients are {" +
            coef[0] + ", " + coef[1] + "}");
    }
}

Output

The computed regression coefficients are {4, 3}

LinearRegression.CoefficientTTestsValue Class

CoefficientTTestsValue contains statistics related to the regression coefficients.

public class Imsl.Stat.LinearRegression.CoefficientTTestsValue

Constructor

CoefficientTTestsValue
    public CoefficientTTestsValue(Imsl.Stat.LinearRegression lr)
    CoefficientTTestsValue contains statistics related to the regression coefficients.

    lr A LinearRegression object used to calculate the regression statistics.
**Methods**

GetCoefficient

```
public double GetCoefficient(int i)

Returns the estimate for a coefficient.
```

- **i** An `int` which specifies the index of the coefficient whose estimate is to be returned.

Returns — A `double` which specifies the estimate for the \( i \)-th coefficient.

GetPValue

```
public double GetPValue(int i)

Returns the p-value for the two-sided test.
```

- **i** An `int` which specifies the index of the coefficient whose \( p \)-value estimate is to be returned.

Returns — A `double` which specifies the estimated \( p \)-value for the \( i \)-th coefficient estimate.

GetStandardError

```
public double GetStandardError(int i)

Returns the estimated standard error for a coefficient estimate.
```

- **i** An `int` which specifies the index of the coefficient whose standard error estimate is to be returned.

Returns — A `double` which specifies the estimated standard error for the \( i \)-th coefficient estimate.

GetTStatistic

```
public double GetTStatistic(int i)

Returns the t-statistic for the test that the \( i \)-th coefficient is zero.
```

- **i** An `int` which specifies the index of the coefficient whose standard error estimate is to be returned.

Returns — A `double` which specifies the estimated standard error for the \( i \)-th coefficient estimate.

---

**NonlinearRegression Class**

Fits a multivariate nonlinear regression model using least squares.

```
public class Imsl.Stat.NonlinearRegression

302 • NonlinearRegression Class  Imsl C# Numerical Library
```
Properties

AbsoluteTolerance

virtual public double AbsoluteTolerance {set; }

The absolute function tolerance. A double scalar value specifying the absolute function tolerance.

The tolerance must be greater than or equal to zero.

The default value is 4.93e-32.

Coefficients

virtual public double[] Coefficients {get; }


DFError

virtual public double DFErro {get; }


Digits

virtual public int Digits {set; }

The number of good digits in the residuals. An int specifying the number of good digits in the residuals.

The number of digits must be greater than zero.

The default value is 15.

ErrorStatus

virtual public int ErrorStatus {get; }

Characterizes the performance of NonlinearRegression. An int specifying information about convergence.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All convergence tests were met.</td>
</tr>
<tr>
<td>1</td>
<td>Scaled step tolerance was satisfied. The current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or StepTolerance is too big.</td>
</tr>
<tr>
<td>2</td>
<td>Scaled actual and predicted reductions in the function are less than or equal to the relative function convergence tolerance RelativeTolerance.</td>
</tr>
<tr>
<td>3</td>
<td>Iterates appear to be converging to a noncritical point. Incorrect gradient information, a discontinuous function, or stopping tolerances being too tight may be the cause.</td>
</tr>
<tr>
<td>4</td>
<td>Five consecutive steps with the maximum stepsize have been taken. Either the function is unbounded below, or has a finite asymptote in some direction, or the MaxStepsize is too small.</td>
</tr>
</tbody>
</table>

Regression NonlinearRegression Class • 303
See Also: RelativeTolerance (p. 305), StepTolerance (p. 305), MaxStepsize (p. 304)

FalseConvergenceTolerance
  
  virtual public double FalseConvergenceTolerance {set; }  
  The false convergence tolerance. A double scalar value specifying the false convergence  
  tolerance.  
  The tolerance must be greater than or equal to zero.  
  The default value is 2.22e-14.

GradientTolerance
  
  virtual public double GradientTolerance {set; }  
  The gradient tolerance. A double specifying the gradient tolerance used to compute the  
  gradient.  
  The tolerance must be greater than or equal to zero.  
  The default value is 6.055e-6.

Guess
  
  virtual public double[] Guess {set; }  
  The initial guess of the parameter values. A double array of initial values for the  
  parameters.  
  The default value is an array of zeroes.

InitialTrustRegion
  
  virtual public double InitialTrustRegion {set; }  
  The initial trust region radius. A double scalar value specifying the initial trust region  
  radius.  
  The initial trust radius must be greater than zero.  
  The default value is set based on the initial scaled Cauchy step.

MaxIterations
  
  virtual public int MaxIterations {set; }  
  The maximum number of iterations allowed during optimization. An int specifying the  
  maximum number of iterations allowed during optimization.  
  The value must be greater than 0.  
  The default value is 100.

MaxStepsize
  
  virtual public double MaxStepsize {set; }  
  The maximum allowable stepsize. A nonnegative double value specifying the maximum  
  allowable stepsize.  
  The maximum allowable stepsize must be greater than zero. If this property is not set  
  then the maximum stepsize is set to a default value based on a scaled theta.

R
  
  virtual public double[,] R {get; }
A copy of the \( R \) matrix. A two dimensional `double` array containing a copy of the \( R \) matrix or `null` if `Imsl.Stat.NonlinearRegression.Solve(Imsl.Stat.NonlinearRegression.IFunction)` (p. 306) has not been called.

The upper triangular matrix containing the \( R \) matrix from a QR decomposition of the matrix of regressors.

**Rank**

```csharp
virtual public int Rank { get; }
```

The rank of the matrix. An `int` which specifies the rank of the matrix or `null` if `Imsl.Stat.NonlinearRegression.Solve(Imsl.Stat.NonlinearRegression.IFunction)` (p. 306) has not been called.

**RelativeTolerance**

```csharp
virtual public double RelativeTolerance { set; }
```

The relative function tolerance. A `double` scalar value specifying the relative function tolerance.

The relative function tolerance must be greater than or equal to zero.

The default value is 1.0e-20.

**Scale**

```csharp
virtual public double[] Scale { set; }
```

The scaling array for \( \theta \). A `double` array containing the scaling values for the parameters (\( \theta \)).

The elements of the scaling array must be greater than zero. `Scale` is used mainly in scaling the gradient and the distance between points. If good starting values of \( \theta \) are known and are nonzero, then a good choice is `Scale[i] = 1.0/\theta[i]`. Otherwise, if \( \theta \) is known to be in the interval \((-10.e5, 10.e5)\), set `Scale[i] = 10.e-5`. By default, the elements of the scaling array are set to 1.0. The default value is an array of ones.

**StepTolerance**

```csharp
virtual public double StepTolerance { set; }
```

The step tolerance. A `double` scalar value specifying the step tolerance used to step between two points.

The step tolerance must be greater than or equal to zero.

The default value is 3.667e-11.

### Constructor

```
NonlinearRegression
public NonlinearRegression(int nparm)
```

Constructs a new nonlinear regression object.

- `nparm` An `int` which specifies the number of unknown parameters in the regression.
Methods

GetCoefficient

virtual public double GetCoefficient(int i)

Returns the estimate for a coefficient.

i An int which specifies the index of a coefficient whose estimate is to be returned.

Returns — A double which contains the estimate for the i-th coefficient or null if
has not been called.

GetSSE

virtual public double GetSSE()

Returns the sums of squares for error.

Returns — A double which contains the sum of squares for error or null if
has not been called.

Solve

virtual public double[] Solve(Imsl.Stat.NonlinearRegression.IFunction F)

Solves the least squares problem and returns the regression coefficients.

F An Imsl.Stat.NonlinearRegression.IFunction (p. 315) whose coefficients are to be
computed.

Returns — A double array containing the regression coefficients.

Imsl.Stat.TooManyIterationsException is thrown when the number of allowed
iterations is exceeded

Imsl.Stat.NegativeFreqException is thrown when the specified frequency is negative

Imsl.Stat.NegativeWeightException is thrown when the weight is negative

Description

The nonlinear regression model is

\[ y_i = f(x_i; \theta) + \varepsilon_i \quad i = 1, 2, \ldots, n \]

where the observed values of the \( y_i \) constitute the responses or values of the dependent variable,
the known \( x_i \) are vectors of values of the independent (explanatory) variables, \( \theta \) is the vector of
\( p \) regression parameters, and the \( \varepsilon_i \) are independently distributed normal errors each with mean
zero and variance \( \sigma^2 \). For this model, a least squares estimate of \( \theta \) is also a maximum likelihood
estimate of \( \theta \).

The residuals for the model are
\[ e_i(\theta) = y_i - f(x_i; \theta) \quad i = 1, 2, \ldots, n \]

A value of \( \theta \) that minimizes

\[ \sum_{i=1}^{n} |e_i(\theta)|^2 \]

is the least-squares estimate of \( \theta \) calculated by this class. \texttt{NonlinearRegression} accepts these residuals one at a time as input from a user-supplied function. This allows \texttt{NonlinearRegression} to handle cases where \( n \) is so large that data cannot reside in an array but must reside in a secondary storage device.

\texttt{NonlinearRegression} is based on MINPACK routines \texttt{LMDIF} and \texttt{LMDER} by More’ et al. (1980). \texttt{NonlinearRegression} uses a modified Levenberg-Marquardt method to generate a sequence of approximations to the solution. Let \( \hat{\theta}_c \) be the current estimate of \( \theta \). A new estimate is given by

\[ \hat{\theta}_c + s_c \]

where \( s_c \) is a solution to

\[ (J(\hat{\theta}_c)^T J(\hat{\theta}_c) + \mu c I)s_c = J(\hat{\theta}_c)^T e(\hat{\theta}_c) \]

Here, \( J(\hat{\theta}_c) \) is the Jacobian evaluated at \( \hat{\theta}_c \).

The algorithm uses a "trust region" approach with a step bound of \( \delta_c \). A solution of the equations is first obtained for \( \mu_c = 0 \). If \( ||s_c||_2 < \delta_c \), this update is accepted; otherwise, \( \mu_c \) is set to a positive value and another solution is obtained. The method is discussed by Levenberg (1944), Marquardt (1963), and Dennis and Schnabel (1983, pages 129 - 147, 218 - 338).

Forward finite differences are used to estimate the Jacobian numerically unless the user supplied function computes the derivatives. In this case the Jacobian is computed analytically via the user-supplied function.

\texttt{NonlinearRegression} does not actually store the Jacobian but uses fast Givens transformations to construct an orthogonal reduction of the Jacobian to upper triangular form. The reduction is based on fast Givens transformations (see Golub and Van Loan 1983, pages 156-162, Gentleman 1974). This method has two main advantages:

- The loss of accuracy resulting from forming the crossproduct matrix used in the equations for \( s_c \) is avoided.
- The \( n \times p \) Jacobian need not be stored saving space when \( n > p \).
A weighted least squares fit can also be performed. This is appropriate when the variance of $\epsilon_i$ in the nonlinear regression model is not constant but instead is $\sigma^2/w_i$. Here, $w_i$ are weights input via the user supplied function. For the weighted case, NonlinearRegression finds the estimate by minimizing a weighted sum of squares error.

**Programming Notes**

Nonlinear regression allows users to specify the model’s functional form. This added flexibility can cause unexpected convergence problems for users who are unaware of the limitations of the algorithm. Also, in many cases, there are possible remedies that may not be immediately obvious. The following is a list of possible convergence problems and some remedies. No one-to-one correspondence exists between the problems and the remedies. Remedies for some problems may also be relevant for the other problems.

- A local minimum is found. Try a different starting value. Good starting values can often be obtained by fitting simpler models. For example, for a nonlinear function

  \[ f(x; \theta) = \theta_1 e^{\theta_2 x} \]

  good starting values can be obtained from the estimated linear regression coefficients $\hat{\beta}_0$ and $\hat{\beta}_1$ from a simple linear regression of $\ln y$ on $\ln x$. The starting values for the nonlinear regression in this case would be

  \[ \theta_1 = e^{\hat{\beta}_0} \text{ and } \theta_2 = \hat{\beta}_1 \]

  If an approximate linear model is unclear, then simplify the model by reducing the number of nonlinear regression parameters. For example, some nonlinear parameters for which good starting values are known could be set to these values. This simplifies the approach to computing starting values for the remaining parameters.

- The estimate of $\theta$ is incorrectly returned as the same or very close to the initial estimate.

  - The scale of the problem may be orders of magnitude smaller than the assumed default of 1 causing premature stopping. For example, if the sums of squares for error is less than approximately $(2.22 e^{-16})^2$, the routine stops. See Example 3, which shows how to shut down some of the stopping criteria that may not be relevant for your particular problem and which also shows how to improve the speed of convergence by the input of the scale of the model parameters.

  - The scale of the problem may be orders of magnitude larger than the assumed default causing premature stopping. The information with regard to the input of the scale of the model parameters in Example 3 is also relevant here. In addition, the maximum allowable step size Imsl.Stat.NonlinearRegression.MaxStepsize (p. 304) in Example 3 may need to be increased.
The residuals are input with accuracy much less than machine accuracy, causing premature stopping because a local minimum is found. Again see Example 3 to see how to change some default tolerances. If you cannot improve the precision of the computations of the residual, you need to use method Imsl.Stat.NonlinearRegression.Digits (p. 303) to indicate the actual number of good digits in the residuals.

- The model is discontinuous as a function of $\theta$. There may be a mistake in the user-supplied function. Note that the function $f(x; \theta)$ can be a discontinuous function of $x$.


- Overflow occurs during the computations. Make sure the user-supplied functions do not overflow at some value of $\theta$.

- The estimate of $\theta$ is going to infinity. A parameterization of the problem in terms of reciprocals may help.

- Some components of $\theta$ are outside known bounds. This can sometimes be handled by making a function that produces artificially large residuals outside of the bounds (even though this introduces a discontinuity in the model function).

Note that the Imsl.Stat.NonlinearRegression.Solve(Imsl.Stat.NonlinearRegression.IFunction) (p. 306) method must be called before using any property as a right operand, otherwise the value is null.

**Example 1: Nonlinear Regression using Finite Differences**

In this example a nonlinear model is fitted. The derivatives are obtained by finite differences.

```csharp
using System;
using Imsl.Math;
using Imsl.Stat;
public class NonlinearRegressionEx1 : NonlinearRegression.IFunction
{
    public bool f(double[] theta, int iobs, double[] frq, double[] wt, double[] e)
    {
        double[] ydata =
            new double[]{ 54.0, 50.0, 45.0, 37.0, 35.0,
                         25.0, 20.0, 16.0, 18.0, 13.0,
                         8.0, 11.0, 8.0, 4.0, 6.0};
        double[] xdata =
            new double[]{ 2.0, 5.0, 7.0, 10.0, 14.0,
                         19.0, 26.0, 31.0, 34.0, 38.0,
                         45.0, 52.0, 53.0, 60.0, 65.0};
        bool iend;
        int nob = 15;
```
if (iobs < nobs)
{
    wt[0] = 1.0;
    frq[0] = 1.0;
    iend = true;
}
else
{
    iend = false;
}
return iend;

public static void Main(String[] args)
{
    int nparm = 2;
    double[] theta = new double[]{60.0, - 0.03};
    NonlinearRegression regression = new NonlinearRegression(nparm);
    regression.Guess = theta;
    NonlinearRegression.IFunction fcn = new NonlinearRegressionEx1();
    double[] coef = regression.Solve(fcn);
    Console.Out.WriteLine("The computed regression coefficients are {" + coef[0] + ", " + coef[1] + "}");
    Console.Out.WriteLine("The computed rank is " + regression.Rank);
    Console.Out.WriteLine("The degrees of freedom for error are " + regression.DFError);
    Console.Out.WriteLine("The sums of squares for error is " + regression.GetSSE());
    new PrintMatrix("R from the QR decomposition ").Print(regression.R);
}

Output

The computed regression coefficients are {58.6065629385189, -0.0395864472964795}
The computed rank is 2
The degrees of freedom for error are 13
The sums of squares for error is 49.4592998624719
   R from the QR decomposition
          0           1
0  1.87385998095046  1139.92835934133
1  0           1139.79755002385

310 • NonlinearRegression Class  IMSL C# Numerical Library
Example 2: Nonlinear Regression with User-supplied Derivatives

In this example a nonlinear model is fitted. The derivatives are supplied by the user.

```csharp
using System;
using Imsl.Math;
using Imsl.Stat;
public class NonlinearRegressionEx2 : NonlinearRegression.IDerivative
{
    double[] ydata = new double[]{
        54.0, 50.0, 45.0, 37.0, 35.0, 25.0, 20.0,
        16.0, 18.0, 13.0, 8.0, 11.0, 8.0, 4.0, 6.0};
    double[] xdata = new double[]{
        2.0, 5.0, 7.0, 10.0, 14.0, 19.0, 26.0, 31.0,
        34.0, 38.0, 45.0, 52.0, 53.0, 60.0, 65.0};
    bool iend;
    int nobs = 15;

    public bool f(double[] theta, int iobs, double[] frq, double[] wt, double[] e)
    {
        if (iobs < nobs)
        {
            wt[0] = 1.0;
            frq[0] = 1.0;
            iend = true;
        }
        else
        {
            iend = false;
        }
        return iend;
    }

    public bool derivative(double[] theta, int iobs, double[] frq, double[] wt, double[] de)
    {
        if (iobs < nobs)
        {
            wt[0] = 1.0;
            frq[0] = 1.0;
            iend = true;
        }
        else
        {
            iend = false;
        }
        return iend;
    }
}
```
```csharp
public static void Main(String[] args)
{
    int nparm = 2;
    double[] theta = new double[]{60.0, -0.03};
    NonlinearRegression regression = new NonlinearRegression(nparm);
    regression.Guess = theta;
    double[] coef = regression.Solve(new NonlinearRegressionEx2());
    Console.Out.WriteLine("The computed regression coefficients are {" +
        coef[0] + ", " + coef[1] + "}");
    Console.Out.WriteLine("The computed rank is " + regression.Rank);
    Console.Out.WriteLine("The degrees of freedom for error are " +
        regression.DFError);
    Console.Out.WriteLine("The sums of squares for error is " +
        regression.GetSSE());
    new PrintMatrix("R from the QR decomposition ").Print(regression.R);
}
```

**Output**

The computed regression coefficients are {58.6065629254192, -0.0395864472775247}
The computed rank is 2
The degrees of freedom for error are 13
The sums of squares for error is 49.4592998624722
R from the QR decomposition

```
0 1
0 1.87385998422826 1139.92837730064
1 1139.79757620697
```

**Example 3: NonlinearRegression using Set Methods**

In this example, some nondefault tolerances and scales are used to fit a nonlinear model. The data is 1.e-10 times the data of Example 1. In order to fit this model without rescaling the data, we first set the absolute function tolerance to 0.0. The default value would cause the program to terminate after one iteration because the residual sum of squares is roughly 1.e-19. We also set the relative function tolerance to 0.0. The gradient tolerance is properly scaled for this problem so we leave it at its default value. Finally, we set the elements of scale to the absolute value of the reciprocal of the starting value. The derivatives are obtained by finite differences.

```csharp
using System;
using Imsl.Math;
using Imsl.Stat;
public class NonlinearRegressionEx3 : NonlinearRegression.IFunction
{
    public bool f(double[] theta, int iobs, double[] frq, double[] wt, double[] e)
    {
```
double[] ydata = new double[]{
  54e-10, 50e-10, 45e-10, 35e-10, 25e-10, 20e-10,
  15e-10, 18e-10, 13e-10, 8e-10, 11e-10, 8e-10, 4e-10, 6e-10};
double[] xdata = new double[]{
  2.0, 5.0, 7.0, 10.0, 14.0, 19.0, 26.0, 31.0, 34.0, 38.0,
  45.0, 52.0, 53.0, 60.0, 65.0};
bool iend;
int nobs = 15;
if (iobs < nobs)
{
  wt[0] = 1.0;
  frq[0] = 1.0;
  iend = true;
}
else
{
  iend = false;
}
return iend;
}
public static void Main(String[] args)
{
  int nparm = 2;
  double[] theta = new double[]{6e-9, -0.03};
  double[] scale = new double[nparm];
  NonlinearRegression regression = new NonlinearRegression(nparm);
  regression.Guess = theta;
  regression.AbsoluteTolerance = 0.0;
  regression.RelativeTolerance = 0.0;
  scale[0] = 1.0 / Math.Abs(theta[0]);
  scale[1] = 1.0 / Math.Abs(theta[1]);
  regression.Scale = scale;
  NonlinearRegression.IFunction fcn = new NonlinearRegressionEx3();
  double[] coef = regression.Solve(fcn);
  Console.Out.WriteLine("The computed regression coefficients are {" +
    coef[0] + ", " + coef[1] + "}");
  Console.Out.WriteLine("The computed rank is "+ regression.Rank);
  Console.Out.WriteLine("The degrees of freedom for error are " + regression.DFError);
  Console.Out.WriteLine("The sums of squares for error is " + regression.GetSSE());
  new PrintMatrix("R from the QR decomposition ").Print(regression.R);
}

Output

The computed regression coefficients are {5.78378362045064E-09, -0.0396252538454606}
The computed rank is 2
The degrees of freedom for error are 13
The sums of squares for error is 5.1663766194061E-19
   R from the QR decomposition
NonlinearRegression.IDerivative Interface

Public interface for the user supplied function to compute the derivative for NonlinearRegression.

public interface Imsl.Stat.NonlinearRegression.IDerivative implements Method

derivative
abstract public bool derivative(double[] theta, int iobs, double[] frq, double[] wt, double[] de)
Computes the weight, frequency, and partial derivatives of the residual given the parameter vector theta for a single observation.
The length of theta corresponds to the number of unknown parameters in the regression function.
The function is evaluated at observation y[iobs].
Use wt = 1.0 for equal weighting (unweighted least squares).
The length of de corresponds to the number of unknown parameters in the regression function.

theta An input double array which contains the parameter values of the regression function.
iobs An input int value indicating the observation index.
frq An output double array of length 1 containing the frequency for observation y[iobs].
wt An output double array of length 1 containing the weight for the observation y[iobs].
de An output double array containing the partial derivatives of the error (residual) for observation y[iobs].

Returns — A boolean value representing the completion indicator. true indicates iobs is less than the number of observations. false indicates iobs is greater than or equal to the number of observations and wt, frq, and de are not output.
NonlinearRegression.IFunction Interface

Public interface for the user supplied function for NonlinearRegression.

```java
public interface Imsl.Stat.NonlinearRegression.IFunction
```

Method

```java
abstract public bool f(double[] theta, int iobs, double[] frq, double[] wt, double[] e)
```

Computes the weight, frequency, and residual given the parameter vector `theta` for a single observation.

The length of `theta` corresponds to the number of unknown parameters in the model.

The function is evaluated at observation `y[iobs]`.

Use `wt = 1.0` for equal weighting (unweighted least squares).

- **theta** An input `double` array containing the parameter values of the model.
- **iobs** An input `int` value indicating the observation index.
- **frq** An output `double` array of length 1 containing the frequency for observation `y[iobs]`.
- **wt** An output `double` array of length 1 containing the weight for observation `y[iobs]`.
- **e** An output `double` array of length 1 which contains the error (residual) for observation `y[iobs]`.

Returns — A `boolean` value representing the completion indicator. `true` indicates `iobs` is less than the number of observations. `false` indicates `iobs` is greater than or equal to the number of observations and `wt`, `freq`, and `e` are not output.

SelectionRegression Class

Selects the best multiple linear regression models.

```java
public class Imsl.Stat.SelectionRegression
```

Properties

- **CriterionOption**
virtual public Imsl.Stat.SelectionRegression.Criterion CriterionOption {get; set; }
The criterion option used to calculate the regression estimates. An int containing the criterion option.

By default for all criteria, subset size \(1, 2, \ldots, k = n_{\text{Candidate}}\) are considered. However, for \(R^2\) the maximum number of subsets can be restricted using property Imsl.Stat.SelectionRegression.MaximumSubsetSize (p. 316).

<table>
<thead>
<tr>
<th>Criterion Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSquared</td>
<td>For (R^2), subset sizes 1, 2, \ldots, \text{MaximumSubsetSize} are examined. This is the default with (\text{MaximumSubsetSize} = n_{\text{Candidate}}).</td>
</tr>
<tr>
<td>AdjustedRSquared</td>
<td>For Adjusted (R^2), subset sizes 1, 2, \ldots, (n_{\text{Candidate}}) are examined.</td>
</tr>
<tr>
<td>MallowsCP</td>
<td>For Mallow’s (C_p), subset sizes 1, 2, \ldots, (n_{\text{Candidate}}) are examined.</td>
</tr>
</tbody>
</table>

See Also: RSquared (p. 326), AdjustedRSquared (p. 326), MallowsCP (p. 326)

MaximumBestFound

virtual public int MaximumBestFound {set; }
The maximum number of best regressions to be found. An int containing the maximum number of best regressions to be reported.

If the \(R^2\) criterion option is selected, the \text{MaximumBestFound} best regressions for each subset size examined are reported. If the adjusted \(R^2\) or Mallow’s \(C_p\) criteria are selected, the \text{MaximumBestFound} among all possible regressions are found. The default value is 1.

See Also: RSquared (p. 326), AdjustedRSquared (p. 326), MallowsCP (p. 326)

MaximumGoodSaved

virtual public int MaximumGoodSaved {set; }
The maximum number of good regressions for each subset size saved. An int containing the maximum number of good regressions saved for each subset size.

\text{MaximumGoodSaved} must be greater than or equal to Imsl.Stat.SelectionRegression.MaximumBestFound (p. 316). Normally, \text{MaximumGoodSaved} should be less than or equal to 10. It should never need be larger than \text{MaximumSubsetSize}, the maximum number of subsets for any subset size.

Computing time required is inversely related to \text{MaximumGoodSaved}. The default value is maximum(10, Imsl.Stat.SelectionRegression.MaximumSubsetSize (p. 316)).

MaximumSubsetSize

virtual public int MaximumSubsetSize {set; }
The maximum subset size if \(R^2\) criterion is used. An int containing the maximum subset size when \(R^2\) criterion is used.

Default: \(\text{MaximumSubsetSize} = n_{\text{Candidate}}\).

See Also: RSquared (p. 326), AdjustedRSquared (p. 326), MallowsCP (p. 326)
Statistics

```csharp
{get; }
A SummaryStatistics object. A SummaryStatistics object containing the Coefficient statistics.
```

**Constructor**

SelectionRegression

```csharp
public SelectionRegression(int nCandidate)
Constructs a new SelectionRegression object.

nCandidate must be greater than 2.

nCandidate An int containing the number of candidate variables (independent variables).
```

**Methods**

Compute

```csharp
virtual public void Compute(double[,] cov, int nObservations)
Computes the best multiple linear regression models using a user-supplied covariance matrix.

cov can be computed using the Imsl.Stat.Covariances (p. 246) class.

cov A double matrix containing a variance-covariance or sum-of-squares and crossproducts matrix, in which the last column must correspond to the dependent variable.

nObservations An int containing the number of observations used to compute cov.

Imsl.Stat.NoVariablesException is thrown if no variables can enter any model
```

Compute

```csharp
virtual public void Compute(double[,] x, double[] y, double[] weights,
double[] frequencies)
Computes the best weighted multiple linear regression models using frequencies for each observation.

The number of columns in x must be equal to the number of variables set in the constructor.

x A double matrix containing the observations of the candidate (independent) variables.

y A double array containing the observations of the dependent variable.

weights A double array containing the weight for each of the observations.

frequencies A double array containing the frequency for each of the observations of x.
```
Imsl.Stat.NoVariablesException is thrown if no variables can enter any model.
Imsl.Stat.NegativeFreqException is thrown if a frequency is less than zero.
Imsl.Stat.NegativeWeightException is thrown if a weight is less than zero.
Imsl.Stat.TooManyObsDeletedException is thrown if more observations have been deleted than were originally entered.
Imsl.Stat.MoreObsDelThanEnteredException is thrown if more observations are being deleted from the output covariance matrix than were originally entered.
Imsl.Stat.DiffObsDeletedException is thrown if different observations are being deleted from the return matrix than were originally entered.

Compute

```csharp
virtual public void Compute(double[,] x, double[] y, double[] weights)
```

Computes the best weighted multiple linear regression models.
The number of columns in x must be equal to the number of variables set in the constructor.

- **x** A `double` matrix containing the observations of the candidate (independent) variables.
- **y** A `double` array containing the observations of the dependent variable.
- **weights** A `double` array containing the weight for each of the observations.

Imsl.Stat.NoVariablesException is thrown if no variables can enter any model.
Imsl.Stat.TooManyObsDeletedException is thrown if more observations have been deleted than were originally entered.
Imsl.Stat.MoreObsDelThanEnteredException is thrown if more observations are being deleted from the output covariance matrix than were originally entered.
Imsl.Stat.DiffObsDeletedException is thrown if different observations are being deleted from the return matrix than were originally entered.

Compute

```csharp
virtual public void Compute(double[,] x, double[] y)
```

Computes the best multiple linear regression models.
The number of columns in x must be equal to the number of variables set in the constructor.

- **x** A `double` matrix containing the observations of the candidate (independent) variables.
- **y** A `double` array containing the observations of the dependent variable.

Imsl.Stat.NoVariablesException is thrown if no variables can enter any model.
Imsl.Stat.TooManyObsDeletedException is thrown if more observations have been deleted than were originally entered.
Imsl.Stat.MoreObsDelThanEnteredException is thrown if more observations are being deleted from the output covariance matrix than were originally entered.
Imsl.Stat.DiffObsDeletedException is thrown if different observations are being deleted from the return matrix than were originally entered.

---

318 • SelectionRegression Class  I MLS C# Numerical Library
Description

Class **SelectionRegression** finds the best subset regressions for a regression problem with three or more independent variables. Typically, the intercept is forced into all models and is not a candidate variable. In this case, a sum-of-squares and crossproducts matrix for the independent and dependent variables corrected for the mean is computed internally. Optionally, **SelectionRegression** supports user-calculated sum-of-squares and crossproducts matrices; see the description of the `Imsl.Stat.SelectionRegression.Compute(System.Double[,],System.Double[])` method.

"Best" is defined by using one of the following three criteria:

- **$R^2$** (in percent)
  
  $$R^2 = 100 \left(1 - \frac{SSE_p}{SST}\right)$$

- **$R^2_a$** (adjusted $R^2$)
  
  $$R^2_a = 100\left[1 - \frac{n - 1}{n - p}\frac{SSE_p}{SST}\right]$$

  Note that maximizing the $R^2_a$ is equivalent to minimizing the residual mean squared error:

  $$\frac{SSE_p}{(n - p)}$$

- **Mallow’s $C_p$ statistic**
  
  $$C_p = \frac{SSE_p}{s_k^2} + 2p - n$$

Here, $n$ is equal to the sum of the frequencies (or the number of rows in $x$ if frequencies are not specified in the `Compute` method), and SST is the total sum-of-squares. $k$ is the number of candidate or independent variables, represented as the `nCandidate` argument in the `SelectionRegression` constructor. SSE$_p$ is the error sum-of-squares in a model containing $p$ regression parameters including $\beta_0$ (or $p - 1$ of the $k$ candidate variables). Variable

$$s_k^2$$

is the error mean square from the model with all $k$ variables in the model. Hocking (1972) and Draper and Smith (1981, pp. 296-302) discuss these criteria.
Class **SelectionRegression** is based on the algorithm of Furnival and Wilson (1974). This algorithm finds the maximum number of good saved candidate regressions for each possible subset size. For more details, see method MaximumGoodSaved. These regressions are used to identify a set of best regressions. In large problems, many regressions are not computed. They may be rejected without computation based on results for other subsets; this yields an efficient technique for considering all possible regressions.

There are cases when the user may want to input the variance-covariance matrix rather than allow it to be calculated. This can be accomplished using the appropriate **Compute** method. Three situations in which the user may want to do this are as follows:

- The intercept is not in the model. A raw (uncorrected) sum of squares and crossproducts matrix for the independent and dependent variables is required. Argument **nObservations** must be set to 1 greater than the number of observations. Form $A^T A$, where $A = [A, Y]$, to compute the raw sum-of-squares and crossproducts matrix.

- An intercept is a candidate variable. A raw (uncorrected) sum of squares and crossproducts matrix for the constant regressor (= 1.0), independent, and dependent variables is required for **cov**. In this case, **cov** contains one additional row and column corresponding to the constant regressor. This row and column contain the sum-of-squares and crossproducts of the constant regressor with the independent and dependent variables. The remaining elements in **cov** are the same as in the previous case. Argument **nObservations** must be set to 1 greater than the number of observations.

- There are $m$ variables that must be forced into the models. A sum-of-squares and crossproducts matrix adjusted for the $m$ variables is required (calculated by regressing the candidate variables on the variables to be forced into the model). Argument **nObservations** must be set to $m$ less than the number of observations.

### Programming Notes

**SelectionRegression** can save considerable CPU time over explicitly computing all possible regressions. However, the function has some limitations that can cause unexpected results for users who are unaware of the limitations of the software.

- For $k + 1 > -\log_2(\epsilon)$, where $\epsilon$ is the largest relative spacing for double precision, some results can be incorrect. This limitation arises because the possible models indicated (the model numbers 1, 2, ..., $2k$) are stored as floating-point values; for sufficiently large $k$, the model numbers cannot be stored exactly. On many computers, this means **SelectionRegression** (for $k > 49$) can produce incorrect results.

- **SelectionRegression** eliminates some subsets of candidate variables by obtaining lower bounds on the error sum-of-squares from fitting larger models. First, the full model containing all independent variables is fit sequentially using a forward stepwise procedure in which one variable enters the model at a time, and criterion values and model numbers for all the candidate variables that can enter at each step are stored. If linearly dependent variables are removed from the full model, a "VariablesDeleted" warning is issued. In this
case, some submodels that contain variables removed from the full model because of linear dependency can be overlooked if they have not already been identified during the initial forward stepwise procedure. If this warning is issued and you want the variables that were removed from the full model to be considered in smaller models, you can rerun the program with a set of linearly independent variables.

Example 1: SelectionRegression

This example uses a data set from Draper and Smith (1981, pp. 629-630). Class SelectionRegression is invoked to find the best regression for each subset size using the $R^2$ criterion.

```csharp
using System;
using Imsl.Math;
using Imsl.Stat;

public class SelectionRegressionEx1
{
    public static void Main(String[] args)
    {
        double[,] x = {{7.0, 26.0, 6.0, 60.0},
                        {1.0, 29.0, 15.0, 52.0},
                        {11.0, 56.0, 8.0, 20.0},
                        {11.0, 31.0, 8.0, 47.0},
                        {7.0, 52.0, 6.0, 33.0},
                        {11.0, 55.0, 9.0, 22.0},
                        {3.0, 71.0, 17.0, 6.0},
                        {1.0, 31.0, 22.0, 44.0},
                        {2.0, 54.0, 18.0, 22.0},
                        {21.0, 47.0, 4.0, 26},
                        {1.0, 40.0, 23.0, 34.0},
                        {11.0, 66.0, 9.0, 12.0},
                        {10.0, 68.0, 8.0, 12.0}};
        double[] y = new double[] { 78.5, 74.3, 104.3, 87.6, 95.9, 109.2, 102.7, 72.5, 93.1, 115.9, 83.8, 113.3, 109.4};
        SelectionRegression sr = new SelectionRegression(4);
        sr.Compute(x, y);
        for (int i = 1; i <= 4; i++)
        {
            double[] tmpCrit = stats.GetCriterionValues(i);
            int[,] indvar = stats.GetIndependentVariables(i);
            Console.Out.WriteLine("Regressions with " + i + " variable(s) (R-squared)");
            for (int j = 0; j < tmpCrit.GetLength(0); j++)
            {
                // Output regression details
            }
        }
    }
}
```
{  
    Console.Out.Write(" " + tmpCrit[j] + " ");
    for (int k = 0; k < indvar.GetLength(1); k++)
        Console.Out.Write(indvar[j,k] + " ");
    Console.Out.WriteLine("\n");
}
Console.Out.WriteLine("\n");

// Setup a PrintMatrix object for use in the loop below.
PrintMatrix pm = new PrintMatrix();
pm.SetColumnSpacing(8);
PrintMatrixFormat tst = new PrintMatrixFormat();
tst.SetNoColumnLabels();
tst.SetNoRowLabels();
tst.NumberFormat = "0.000";
for (int i = 0; i < 4; i++)
{
    double[,] tmpCoef = stats.GetCoefficientStatistics(i);
    Console.Out.WriteLine("Regressions with "+(i+1)+" variable(s) (R-squared)\n");  
    Console.Out.WriteLine(  "Variable Coefficient Standard Error t-statistic p-value\n" );
    pm.Print(tst, tmpCoef);
}
}

Output

Regressions with 1 variable(s) (R-squared)
67.4541964131609 4
66.6268257633294 2
53.3948023835034 1
28.5872731229812 3

Regressions with 2 variable(s) (R-squared)
97.8678374535632 1 2
97.2471047716931 1 4
93.5289640615808 3 4
68.00640795005 2 4
54.8166748844824 1 3

Regressions with 3 variable(s) (R-squared)
98.2335451200427 1 2 4
98.2284679219087 1 2 3
98.1281092587344 1 3 4
97.2819959386273 2 3 4

Regressions with 4 variable(s) (R-squared)
98.237562040768 1 2 3 4
Example 2: SelectionRegression

This example uses the same data set as the first example, but Mallow’s $C_p$ statistic is used as the criterion rather than $R^2$. Note that when Mallow’s $C_p$ statistic (or adjusted $R^2$) is specified, **MaximumBestFound** is used to indicate the total number of ”best” regressions (rather than indicating the number of best regressions per subset size, as in the case of the $R^2$ criterion). In this example, the three best regressions are found to be (1, 2), (1, 2, 4), and (1, 2, 3).

```csharp
using System;
using Imsl.Math;
using Imsl.Stat;

public class SelectionRegressionEx2
{
    public static void Main(String[] args)
    {
        double[,] x = {
            {7.0, 26.0, 6.0, 60.0},
            {1.0, 29.0, 15.0, 52.0},
            {11.0, 56.0, 8.0, 20.0},
            {11.0, 31.0, 8.0, 47.0},
            {7.0, 52.0, 6.0, 33.0},
        };
double[] y = new double[]{ 78.5, 74.3, 104.3, 87.6, 95.9, 109.2, 102.7, 72.5, 93.1, 115.9, 83.8, 113.3, 109.4};

SelectionRegression sr = new SelectionRegression(4);
sr.MaximumBestFound = 3;
sr.Compute(x, y);

for (int i = 1; i <= 4; i++)
{
    double[] tmpCrit = stats.GetCriterionValues(i);
    int[,] indvar = stats.GetIndependentVariables(i);
    Console.Out.WriteLine("Regressions with "+i+" variable(s) (MallowsCP)");
    for (int j = 0; j < tmpCrit.GetLength(0); j++)
    {
        Console.Out.Write(" " + tmpCrit[j] + " ");
        for (int k = 0; k < indvar.GetLength(1); k++)
        {
            Console.Out.Write(indvar[j,k] + " ");
        }
        Console.Out.WriteLine("");
    }
    Console.Out.WriteLine("");
}

// Setup a PrintMatrix object for use in the loop below.
PrintMatrix pm = new PrintMatrix();
pm.SetColumnSpacing(9);
PrintMatrixFormat tst = new PrintMatrixFormat();
tst.SetNoColumnLabels();
tst.SetNoRowLabels();
tst.NumberFormat = "0.000";
for (int i = 0; i < 3; i++)
{
    double[,] tmpCoef = stats.GetCoefficientStatistics(i);
    Console.Out.WriteLine("Regressions with "+(i+1)+" variable(s) (MallowsCP)";
    Console.Out.WriteLine("Variable Coefficient Standard Error t-statistic p-value";
    pm.Print(tst, tmpCoef);
}
Output

Regressions with 1 variable(s) (MallowsCP)
138.730833491679 4
142.486406936963 2
202.548769123452 1
315.154284140084 3

Regressions with 2 variable(s) (MallowsCP)
2.67824159831843 1 2
5.49585082475865 1 4
22.3731119646976 3 4
138.225919754643 2 4
198.094652569591 1 3

Regressions with 3 variable(s) (MallowsCP)
3.0182347348735 1 2 4
3.04127972306417 1 2 3
3.4968244324848 1 3 4
7.33747399565598 2 3 4

Regressions with 4 variable(s) (MallowsCP)
5 1 2 3 4

Regressions with 1 variable(s) (MallowsCP)
Variable Coefficient Standard Error t-statistic p-value
1.000 1.468 0.121 12.105 0.000
2.000 0.662 0.046 14.442 0.000

Regressions with 2 variable(s) (MallowsCP)
Variable Coefficient Standard Error t-statistic p-value
1.000 1.452 0.117 12.410 0.000
2.000 0.416 0.186 2.242 0.052
4.000 -0.237 0.173 -1.365 0.205

Regressions with 3 variable(s) (MallowsCP)
Variable Coefficient Standard Error t-statistic p-value
1.000 1.696 0.205 8.290 0.000
2.000 0.657 0.044 14.851 0.000
3.000 0.250 0.185 1.354 0.209
SelectionRegression.Criterion Enumeration

Criterion Methods.

public enumeration Imsl.Stat.SelectionRegression.Criterion

Fields

AdjustedRSquared
  Indicates $R^2_a$ (adjusted $R^2$) criterion regression.

MallowsCP
  public Imsl.Stat.SelectionRegression.Criterion MallowsCP
  Indicates Mallow’s $C_p$ criterion regression.

RSquared
  public Imsl.Stat.SelectionRegression.Criterion RSquared
  Indicates $R^2$ criterion regression.

SelectionRegression.SummaryStatistics Class

SummaryStatistics contains statistics related to the regression coefficients.

public class Imsl.Stat.SelectionRegression.SummaryStatistics

Methods

GetCoefficientStatistics
  virtual public double[,] GetCoefficientStatistics(int regressionIndex)
  Returns the coefficients statistics for each of the best regressions found for each subset
  considered.

  The value set using Imsl.Stat.SelectionRegression.MaximumBestFound (p. 316) determines
  the total number of best regressions to find. The number of best regression is equal to
  (Imsl.Stat.SelectionRegression.MaximumSubsetSize (p. 316) x MaximumBestFound), if
  criterion RSquared is specified or it is equal to MaximumBestFound if either
  MallowsCP or AdjustedRSquared is specified.

  Each row contains statistics related to the regression coefficients of the best models. The
  regressions are ordered so that the better regressions appear first. The statistic in the
  columns are as follows (inferences are conditional on the selected model):
### StepwiseRegression Class

Builds multiple linear regression models using forward selection, backward selection, or stepwise selection.

```java
public class Imsl.Stat.StepwiseRegression
```

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>variable number</td>
</tr>
<tr>
<td>1</td>
<td>coefficient estimate</td>
</tr>
<tr>
<td>2</td>
<td>estimated standard error of the estimate</td>
</tr>
<tr>
<td>3</td>
<td>t-statistic for the test that the coefficient is 0</td>
</tr>
<tr>
<td>4</td>
<td>p-value for the two-sided t test</td>
</tr>
</tbody>
</table>

There will be 0 to \((\text{MaximumSubsetSize} \times \text{MaximumBestFound} - 1)\) best regressions if `RSquared` is specified or 0 to \((\text{MaximumBestFound} - 1)\) if either `MallowsCP` or `AdjustedRSquared` is specified.

See Also:  `RSquared (p. 326), AdjustedRSquared (p. 326), MallowsCP (p. 326)`

**regressionIndex**  An int which specifies the index of the best regression statistics to return.

**Returns** — A two-dimensional double array containing the regression statistics.

```java
GetCriterionValues
```

```java
virtual public double[] GetCriterionValues(int numVariables)
Returns an array containing the values of the best criterion for the number of variables considered.
```

**numVariables**  An int which specifies the number of variables considered.

**Returns** — A double array with `Imsl.Stat.SelectionRegression.MaximumSubsetSize (p. 316)` rows and `nCandidate` columns containing the criterion values.

```java
GetIndependentVariables
```

```java
virtual public int[,] GetIndependentVariables(int numVariables)
Returns the identification numbers for the independent variables for the number of variables considered and in the same order as the criteria returned by `Imsl.Stat.SelectionRegression.SummaryStatistics.GetCriterionValues(System.Int32) (p. 327).`
```

**numVariables**  An int which specifies the number of variables considered.

**Returns** — An int array containing the identification numbers for the independent variables considered.
Properties

ANOVA

virtual public Imsl.Stat.ANOVA ANOVA {get; }

CoefficientTTests

virtual public Imsl.Stat.StepwiseRegression.CoefficientTTestsValue CoefficientTTests {get; }
The student-t test statistics for the regression coefficients. A
StepwiseRegression.CoefficientTTestsValue object containing statistics relating to the regression coefficients.

CoefficientVIF

virtual public double[] CoefficientVIF {get; }
The variance inflation factors for the final model in this invocation. A double array containing the variance inflation factors for the final model in this invocation.
The elements are in the same order as the independent variables in x (or, if the covariance matrix is specified, the elements are in the same order as the variables in cov). Each element corresponding to a variable not in the model contains statistics for a model which includes the variables of the final model and the variables corresponding to the element in question.
The square of the multiple correlation coefficient for the i-th regressor after all others can be obtained from the i-th element for the returned array by the following formula:

\[ 1.0 - \frac{1.0}{VIF} \]

CovariancesSwept

virtual public double[,] CovariancesSwept {get; }
Results after cov has been swept for the columns corresponding to the variables in the model. A double matrix containing the results after cov has been swept on the columns corresponding to the variables in the model.
The estimated variance-covariance matrix of the estimated regression coefficients in the final model can be obtained by extracting the rows and columns corresponding to the independent variables in the final model and multiplying the elements of this matrix by the error mean square.

Force

virtual public int Force {set; }
Forces independent variables into the model based on their level assigned from Levels.
An int specifying the upper bound on the variables forced into the model.
Variables with levels 1, 2, ..., Force are forced into the model as independent variables.
See Also:  Levels (p. 329)
History

```csharp
virtual public double[] History {get; }
```

The stepwise regression history for the independent variables. A `double` array containing the recent history of the independent variables.

The last element corresponds to the dependent variable.

<table>
<thead>
<tr>
<th>History[i]</th>
<th>Status of i-th Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>This variable has never been added to the model.</td>
</tr>
<tr>
<td>0.5</td>
<td>This variable was added into the model during initialization.</td>
</tr>
<tr>
<td>( k &gt; 0.0 )</td>
<td>This variable was added to the model during the ( k )-th step.</td>
</tr>
<tr>
<td>( k &lt; 0.0 )</td>
<td>This variable was deleted from model during the ( k )-th step</td>
</tr>
</tbody>
</table>

See Also: Levels (p. 329)

Levels

```csharp
virtual public int[] Levels {set; }
```

The levels of priority for variables entering and leaving the regression. An `int` array containing the levels of entry into the model for each variable.

Each variable is assigned a positive value which indicates its level of entry into the model. A variable can enter the model only after all variables with smaller nonzero levels of entry have entered. Similarly, a variable can only leave the model after all variables with higher levels of entry have left. Variables with the same level of entry compete for entry (deletion) at each step. A value `Levels[i]=0` means the `i`-th variable never enters the model. A value `Levels[i]=-1` means the `i`-th variable is the dependent variable. The last element in `Levels` must correspond to the dependent variable, except when the variance-covariance or sum-of-squares and crossproducts matrix is supplied.

Default: 1, 1, ..., 1, -1 where -1 corresponds to the dependent variable.

See Also: Force (p. 328)

Method

```csharp
virtual public Imsl.Stat.StepwiseRegression.Direction Method {set; }
```

Specifies the stepwise selection method, forward, backward, or stepwise Regression. An `int` value between -1 and 1 specifying the stepwise selection method.

Fields `Forward`, `Backward`, and `Stepwise` should be used.

Default: `Direction.Stepwise`.

See Also: Forward (p. 336), Backward (p. 336), Stepwise (p. 336)

PValueIn

```csharp
virtual public double PValueIn {set; }
```

Defines the largest p-value for variables entering the model. A `double` containing the largest p-value for variables entering the model.

Variables with p-value less than `PValueIn` may enter the model. Backward regression does not use this value.

Default: `PValueIn = 0.05`. 
PValueOut

    virtual public double PValueOut {set; }
Defines the smallest $p$-value for removing variables. A double containing the smallest
$p$-value for removing variables from the model.
Variables with $p$-values greater than $PValueOut$ may leave the model. $PValueOut$ must be
greater than or equal to $PValueIn$. A common choice for $PValueOut$ is $2 \times PValueIn$.
Forward regression does not use this value.
Default: $PValueOut = 0.10$.

Swept

    virtual public double[] Swept {get; }
An array containing information indicating whether or not a particular variable is in the
model. A double array with information to indicate the independent variables in the
model.
The last element corresponds to the dependent variable. A +1 in the $i$-th position
indicates that the variable is in the selected model. A -1 indicates that the variable is not
in the selected model.
See Also: Levels (p. 329)

Tolerance

    virtual public double Tolerance {set; }
The tolerance used to detect linear dependence among the independent variables. A
double containing the tolerance used for detecting linear dependence.
Default: $Tolerance = 2.2204460492503e-16$.

Constructors

StepwiseRegression

    public StepwiseRegression(double[,] x, double[] y)
Creates a new instance of StepwiseRegression.
    x A double matrix of $nObs$ by $nVars$, where $nObs$ is the number of observations and
    $nVars$ is the number of independent variables.
    y A double array containing the observations of the dependent variable.

Imsl.Stat.TooManyObsDeletedException is thrown if more observations have been
deleted than were originally entered
Imsl.Stat.MoreObsDelThanEnteredException is thrown if more observations are being
deleted from the output covariance matrix than were originally entered
Imsl.Stat.DiffObsDeletedException is thrown if different observations are being
deleted from the return matrix than were originally entered

StepwiseRegression

    public StepwiseRegression(double[,] x, double[] y, double[] weights)
Creates a new instance of weighted StepwiseRegression.
x  A double matrix of nObs by nVars, where nObs is the number of observations and nVars is the number of independent variables.
y  A double array containing the observations of the dependent variable.
weights  A double array containing the weight for each observation of x.

Imsl.Stat.TooManyObsDeletedException is thrown if more observations have been deleted than were originally entered
Imsl.Stat.MoreObsDelThanEnteredException is thrown if more observations are being deleted from the output covariance matrix than were originally entered
Imsl.Stat.DiffObsDeletedException is thrown if different observations are being deleted from the return matrix than were originally entered
Imsl.Stat.NegativeWeightException is thrown if a weight is less than zero.

StepwiseRegression
public StepwiseRegression(double[,] x, double[] y, double[] weights, double[] frequencies)
Creates a new instance of weighted StepwiseRegression using observation frequencies.

x  A double matrix of nObs by nVars, where nObs is the number of observations and nVars is the number of independent variables.
y  A double array containing the observations of the dependent variable.
weights  A double array containing the weight for each observation of x.
frequencies  A double array containing the frequency for each row of x.

Imsl.Stat.TooManyObsDeletedException is thrown if more observations have been deleted than were originally entered
Imsl.Stat.MoreObsDelThanEnteredException is thrown if more observations are being deleted from the output covariance matrix than were originally entered
Imsl.Stat.DiffObsDeletedException is thrown if different observations are being deleted from the return matrix than were originally entered
Imsl.Stat.NegativeWeightException is thrown if a weight is less than zero.
Imsl.Stat.NegativeFreqException is thrown if a frequency is less than zero.

StepwiseRegression
public StepwiseRegression(double[,] cov, int nObservations)
Creates a new instance of StepwiseRegression from a user-supplied variance-covariance matrix.
cov can be computed using the Imsl.Stat.Covariances (p. 246) class.
cov   A double matrix containing a variance-covariance or sum-of-squares and crossproducts matrix, in which the last column must correspond to the dependent variable.
nObservations   An int containing the number of observations associated with cov.
Method

Compute

```csharp
virtual public void Compute()
Builds the multiple linear regression models using forward selection, backward selection,
or stepwise selection.

Imsl.Stat.NoVariablesEnteredException is thrown if no variables entered the model.
All elements of the Imsl.Stat.StepwiseRegression.ANOVA (p. 328) table are set to NaN.
Imsl.Stat.CyclingIsOccurringException is thrown if cycling occurs.
```

Description

Class StepwiseRegression builds a multiple linear regression model using forward selection, backward selection, or forward stepwise (with a backward glance) selection.

Levels of priority can be assigned to the candidate independent variables using
Imsl.Stat.StepwiseRegression.Levels (p. 329). All variables with a priority level of 1 must enter
the model before variables with a priority level of 2. Similarly, variables with a level of 2 must
enter before variables with a level of 3, etc. Variables also can be forced into the model using
Imsl.Stat.StepwiseRegression.Force (p. 328). Note that specifying "force" without also
specifying levels of priority will result in all variables being forced into the model.

Typically, the intercept is forced into all models and is not a candidate variable. In this case, a
sum-of-squares and crossproducts matrix for the independent and dependent variables
corrected for the mean is required. Other possibilities are as follows:

- The intercept is not in the model. A raw (uncorrected) sum-of-squares and crossproducts
  matrix for the independent and dependent variables is required as input in cov. Argument
  nObservations must be set to one greater than the number of observations.

- An intercept is a candidate variable. A raw (uncorrected) sum-of-squares and
  crossproducts matrix for the constant regressor (=1), independent and dependent
  variables are required for cov. In this case, cov contains one additional row and column
  corresponding to the constant regressor. This row/column contains the sum-of-squares
  and crossproducts of the constant regressor with the independent and dependent
  variables. The remaining elements in cov are the same as in the previous case. Argument
  nObservations must be set to one greater than the number of observations.

The stepwise regression algorithm is due to Efroymson (1960). StepwiseRegression uses
sweeps of the covariance matrix (input in cov, if the covariance matrix is specified, or generated
internally) to move variables in and out of the model (Hemmerle 1967, Chapter 3). The
SWEEP operator discussed in Goodnight (1979) is used. A description of the stepwise
algorithm is also given by Kennedy and Gentle (1980, pp. 335-340). The advantage of stepwise
model building over all possible regression (SelectionRegression) is that it is less demanding
computationally when the number of candidate independent variables is very large. However,
there is no guarantee that the model selected will be the best model (highest $R^2$) for any subset size of independent variables.

**Example: StepwiseRegression**

This example uses a data set from Draper and Smith (1981, pp. 629-630). Method `Compute` is invoked to find the best regression for each subset size using the $R^2$ criterion. By default, stepwise regression is performed.

```csharp
using System;
using Imsl.Math;
using Imsl.Stat;

public class StepwiseRegressionEx1
{
    public static void Main(String[] args)
    {
        double[,] x = {
            {7.0, 26.0, 6.0, 60.0},
            {1.0, 29.0, 15.0, 52.0},
            {11.0, 56.0, 8.0, 20.0},
            {11.0, 31.0, 8.0, 47.0},
            {7.0, 52.0, 6.0, 33.0},
            {11.0, 55.0, 9.0, 22.0},
            {3.0, 71.0, 17.0, 6.0},
            {1.0, 31.0, 22.0, 44.0},
            {2.0, 54.0, 18.0, 22.0},
            {21.0, 47.0, 4.0, 26},
            {1.0, 40.0, 23.0, 34.0},
            {11.0, 66.0, 9.0, 12.0},
            {10.0, 68.0, 8.0, 12.0}};
        double[] y = new double[]{78.5, 74.3, 104.3, 87.6,
            95.9, 109.2, 102.7, 72.5,
            93.1, 115.9, 83.8, 113.3, 109.4};

        StepwiseRegression sr = new StepwiseRegression(x, y);
        sr.Compute();

        PrintMatrix pm = new PrintMatrix();
        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.NumberFormat = "0.000";
        pm.SetTitle("*** ANOVA ***");
        pm.Print(sr.ANOVA.GetArray());

        StepwiseRegression.CoefficientTTestsValue coefT = sr.CoefficientTTests;
        double[,] coef = new double[4,4];
        for (int i = 0; i < 4; i++)
        {
            coef[i,0] = coefT.GetCoefficient(i);
            coef[i,1] = coefT.GetStandardError(i);
            coef[i,2] = coefT.GetTStatistic(i);
            coef[i,3] = coefT.GetPValue(i);
        }
    }
}
```
Output

*** ANOVA ***

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>2641.00096476634</td>
</tr>
<tr>
<td>4</td>
<td>74.7621121567348</td>
</tr>
<tr>
<td>5</td>
<td>2715.76307692308</td>
</tr>
<tr>
<td>6</td>
<td>1320.50048238317</td>
</tr>
<tr>
<td>7</td>
<td>7.47621121567348</td>
</tr>
<tr>
<td>8</td>
<td>176.62696308189</td>
</tr>
<tr>
<td>9</td>
<td>1.58106022718485E-08</td>
</tr>
<tr>
<td>10</td>
<td>97.2471047716931</td>
</tr>
<tr>
<td>11</td>
<td>96.6965257260318</td>
</tr>
<tr>
<td>12</td>
<td>2.73426612012684</td>
</tr>
<tr>
<td>13</td>
<td>NaN</td>
</tr>
<tr>
<td>14</td>
<td>NaN</td>
</tr>
</tbody>
</table>

*** Coef ***

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.440</td>
<td>0.138</td>
<td>10.403</td>
</tr>
<tr>
<td>1</td>
<td>0.416</td>
<td>0.186</td>
<td>2.242</td>
</tr>
<tr>
<td>2</td>
<td>-0.410</td>
<td>0.199</td>
<td>-2.058</td>
</tr>
<tr>
<td>3</td>
<td>-0.614</td>
<td>0.049</td>
<td>-12.621</td>
</tr>
</tbody>
</table>

*** Swept ***

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
</tbody>
</table>

*** History ***

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

*** VIF ***

<table>
<thead>
<tr>
<th>0</th>
<th>334</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>StepwiseRegression Class IMSL C# Numerical Library</td>
</tr>
</tbody>
</table>
0  1.0641052101769
1  18.780308640958
2  3.45960147891528
3  1.0641052101769

*** CovS ***

0   1   2   3   4
0  0.003 -0.029 -0.946 0.000 1.440
1 -0.029 154.720 -142.800 0.907 64.381
2 -0.946 -142.800 142.302 0.070 -58.350
3  0.000  0.907  0.070  0.000   -0.614
4  1.440  64.381 -58.350 -0.614  74.762

---

**StepwiseRegression.CoefficientTTestsValue Class**

`CoefficientTTestsValue` contains statistics related to the student-t test, for each regression coefficient.

```java
public class Imsl.Stat.StepwiseRegression.CoefficientTTestsValue

Methods

GetCoefficient

virtual public double GetCoefficient(int index)

Returns the estimate for a coefficient of the independent variable.

`index` must be between 1 and the number of independent variables.

`index`  An int which specifies the index of the coefficient whose estimate is to be returned.

Returns — A double which contains the estimate for the coefficient.

GetPValue

virtual public double GetPValue(int index)

Returns the p-value for the two-sided test $H_0: \beta = 0$ vs. $H_1: \beta \neq 0$.

`index` must be between 1 and the number of independent variables.

`index`  An int which specifies the index of the coefficient whose p-value is to be returned.

Returns — A double which contains the estimated p-value for the coefficient.

GetStandardError

virtual public double GetStandardError(int index)

Returns the estimated standard error for a coefficient estimate.
```
index must be between 1 and the number of independent variables.

index  An int which specifies the index of the coefficient whose standard error estimate is to be returned.

Returns — A double which contains the estimated standard error for the coefficient.

GetTStatistic

virtual public double GetTStatistic(int index)
Returns the student-\( t \) test statistic for testing the \( i \)-th coefficient equal to zero (\( \beta_{index} = 0 \)).

index must be between 1 and the number of independent variables.

index  An int which specifies the index of the coefficient whose \( t \)-test statistic is to be returned.

Returns — A double which contains the estimated \( t \)-test statistic for the coefficient.

---

StepwiseRegression.Direction Enumeration

Direction indicator.

public enumeration Imsl.Stat.StepwiseRegression.Direction

Fields

Backward

public Imsl.Stat.StepwiseRegression.Direction Backward
Indicates backward regression. An attempt is made to remove a variable from the model. A variable is removed if its \( p \)-value exceeds \( PValueOut \). During initialization, all candidate independent variables enter the model.

Forward

public Imsl.Stat.StepwiseRegression.Direction Forward
Indicates forward regression. An attempt is made to add a variable to the model. A variable is added if its \( p \)-value is less than \( PValueIn \). During initialization, only forced variables enter the model.

Stepwise

public Imsl.Stat.StepwiseRegression.Direction Stepwise
Indicates stepwise regression. A backward step is attempted. After the backward step, a forward step is attempted. This is a stepwise step. Any forced variables enter the model during initialization.
UserBasisRegression Class

Generates summary statistics using user-supplied functions in a nonlinear regression model.

public class Imsl.Stat.UserBasisRegression

Property

ANOVA

public Imsl.Stat.ANOVA ANOVA {get; }

Constructor

UserBasisRegression

public UserBasisRegression(Imsl.Stat.IRegressionBasis basis, int nBasis, bool hasIntercept)
Constructs a UserBasisRegression object.

basis A IRegressionBasis basis function supplied by the user.
nBasis A int which specifies the number of basis functions.
hasIntercept A boolean which specifies whether or not the model has an intercept.

Methods

GetCoefficients

public double[] GetCoefficients()
Returns the regression coefficients.

If hasIntercept is false its length is equal to the number of variables. If hasIntercept is true then its length is the number of variables plus one and the 0-th entry is the value of the intercept.

Returns — A double array containing the regression coefficients.

Imsl.Math.SingularMatrixException is thrown when the regression matrix is singular

Update

public void Update(double x, double y, double w)
Adds a new observation and associated weight to the IRegressionBasis object.

x A double containing the independent (explanatory) variable.
y A double containing the dependent (response) variable.
w A double representing the weight.
Example: Regression with User-supplied Basis Functions

In this example, we fit the function $1 + \sin(x) + 7 \cdot \sin(3x)$ with no error introduced. The function is evaluated at 90 equally spaced points on the interval $[0, 6]$. Four basis functions are used, $\sin(kx)$ for $k = 1,\ldots,4$ with no intercept.

```csharp
using System;
using Imsl.Stat;
using Imsl.Math;

public class UserBasisRegressionEx1 : IRegressionBasis
{
    public double Basis(int index, double x)
    {
        return System.Math.Sin((index + 1) * x);
    }

    public static void Main(String[] args)
    {
        double[] coef = new double[4];
        IRegressionBasis basis = new UserBasisRegressionEx1();
        UserBasisRegression ubr =
            new UserBasisRegression(basis, 4, false);
        for (int k = 0; k < 90; k++)
        {
            double x = 6.0 * k / 89.0;
            double y = 1.0 + Math.Sin(x) + 7.0 * Math.Sin(3.0 * x);
            ubr.Update(x, y, 1.0);
        }
        coef = ubr.GetCoefficients();
        new PrintMatrix
            ("The regression coefficients are:").Print(coef);
    }
}
```

Output

The regression coefficients are:

0 0 1.01010532376649
1 0.0199013147736359
2 7.02909074858517
3 0.0374000977854434
IRegressionBasis Interface

Interface for user supplied function to UserBasisRegression object.

public interface Imsl.Stat.IRegressionBasis

Method

Basis

abstract public double Basis(int index, double x)
Basis function for the nonlinear least-squares function.

index  A int which specifies the index of the basis function to be evaluated at x.
x  A double which specifies the point at which the function is to be evaluated.

Returns — A double which specifies the returned value of the function at x.
Chapter 14: Analysis of Variance

Types

class ANOVA ................................................................. 341
class ANOVAFactorial ...................................................... 346
enumeration ANOVAFactorial.ErrorCalculation ....................... 356
class MultipleComparisons ............................................. 357

Usage Notes

The classes described in this chapter are for commonly-used experimental designs. Typically, responses are stored in the input vector $y$ in a pattern that takes advantage of the balanced design structure. Consequently, the full set of model subscripts is not needed to identify each response. The classes assume the usual pattern, which requires that the last model subscript change most rapidly, followed by the model subscript next in line, and so forth, with the first subscript changing at the slowest rate. This pattern is referred to as lexicographical ordering.

ANOVA class allows missing responses if confidence interval information is not requested. Double.NaN (Not a Number) is the missing value code used by these classes. Any element of $y$ that is missing must be set to NaN. Other classes described in this chapter do not allow missing responses because the classes generally deal with balanced designs.

As a diagnostic tool for determination of the validity of a model, classes in this chapter typically perform a test for lack of fit when $n(n > 1)$ responses are available in each cell of the experimental design.

ANOVA Class

Analysis of Variance table and related statistics.
public class Imsl.Stat.ANOVA

Properties

AdjustedRSquared
   public double AdjustedRSquared {get; }
   Returns the adjusted R-squared (in percent). A double representing the adjusted R-squared (in percent).

CoefficientOfVariation
   public double CoefficientOfVariation {get; }
   Returns the coefficient of variation (in percent). A double representing the coefficient of variation (in percent).

DegreesOfFreedomForError
   public double DegreesOfFreedomForError {get; }
   Returns the degrees of freedom for error. A double representing the degrees of freedom for the error.

DegreesOfFreedomForModel
   public double DegreesOfFreedomForModel {get; }
   Returns the degrees of freedom for the model. A double representing the degrees of freedom for the model.

ErrorMeanSquare
   public double ErrorMeanSquare {get; }
   Returns the error mean square. A double representing the error mean square.

F
   public double F {get; }
   Returns the F statistic. A double representing the F statistic.

MeanOfY
   public double MeanOfY {get; }
   Returns the mean of the response (dependent variable). A double representing the mean of the response (dependent variable).

ModelErrorStdev
   public double ModelErrorStdev {get; }
   Returns the estimated standard deviation of the model error. A double representing the estimated standard deviation of the model error.

ModelMeanSquare
   public double ModelMeanSquare {get; }
   Returns the model mean square. A double representing the model mean square.

P
   public double P {get; }
   Returns the p-value. A double representing the p-value.
RSquared
    public double RSquared {get; }
    Returns the $R$-squared (in percent). A double representing the $R$-squared (in percent).

SumOfSquaresForError
    public double SumOfSquaresForError {get; }
    Returns the sum of squares for error. A double representing the sum of squares for the error.

SumOfSquaresForModel
    public double SumOfSquaresForModel {get; }
    Returns the sum of squares for model. A double representing the sum of squares for the model.

TotalDegreesOfFreedom
    public double TotalDegreesOfFreedom {get; }
    Returns the total degrees of freedom. A double representing the total degrees of freedom.

TotalMissing
    public int TotalMissing {get; }
    Returns the total number of missing values. An int representing the total number of missing values ($NaN$) in input $Y$.
    Elements of $Y$ containing $NaN$ (not a number) are omitted from the computations.

TotalSumOfSquares
    public double TotalSumOfSquares {get; }
    Returns the total sum of squares. A double representing the total sum of squares.

Constructors

ANOVA
    public ANOVA(double[,] y)
    Analyzes a one-way classification model.
    The rows in $y$ correspond to observation groups. Each row of $y$ can contain a different number of observations.

    $y$ Two-dimension double array containing the responses.

ANOVA
    public ANOVA(double dfr, double ssr, double dfe, double sse, double gmean)
    Construct an analysis of variance table and related statistics. Intended for use by the LinearRegression class.
    If the grand mean is not known it may be set to not-a-number.

    dfr A double representing the degrees of freedom for the model.
    ssr A double representing the sum of squares for the model.
**Methods**

**GetArray**

```csharp
public double[] GetArray()

Returns the ANOVA values as an array.

Returns — A `double[15]` array containing the following values.
```

<table>
<thead>
<tr>
<th>index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Degrees of freedom for model</td>
</tr>
<tr>
<td>1</td>
<td>Degrees of freedom for error</td>
</tr>
<tr>
<td>2</td>
<td>Total degrees of freedom</td>
</tr>
<tr>
<td>3</td>
<td>Sum of squares for model</td>
</tr>
<tr>
<td>4</td>
<td>Sum of squares for error</td>
</tr>
<tr>
<td>5</td>
<td>Total sum of squares</td>
</tr>
<tr>
<td>6</td>
<td>Model mean square</td>
</tr>
<tr>
<td>7</td>
<td>Error mean square</td>
</tr>
<tr>
<td>8</td>
<td>F statistic</td>
</tr>
<tr>
<td>9</td>
<td>p-value</td>
</tr>
<tr>
<td>10</td>
<td>R-squared (in percent)</td>
</tr>
<tr>
<td>11</td>
<td>Adjusted R-squared (in percent)</td>
</tr>
<tr>
<td>12</td>
<td>Estimated standard deviation of the model error</td>
</tr>
<tr>
<td>13</td>
<td>Mean of the response (dependent variable)</td>
</tr>
<tr>
<td>14</td>
<td>Coefficient of variation (in percent)</td>
</tr>
</tbody>
</table>
```

**GetDunnSidak**

```csharp
public double GetDunnSidak(int i, int j)

Computes the confidence intervals on `i`-th mean - `j`-th mean using the Dunn-Sidak method.

`i` An `int` indicating the `i`-th mean, \( \mu_i \).

`j` An `int` containing the `j`-th mean \( \mu_j \).

Returns — The confidence intervals on `i`-th mean - `j`-th mean using the Dunn-Sidak method.
```

**GetGroupInformation**

```csharp
public double[,] GetGroupInformation()

Returns information concerning the groups.

Row `i` contains information pertaining to the `i`-th group. The information in the columns is as follows:
```
Returns — A two-dimensional double array containing information concerning the groups.

Example: ANOVA

This example computes a one-way analysis of variance for data discussed by Searle (1971, Table 5.1, pages 165-179). The responses are plant weights for 6 plants of 3 different types - 3 normal, 2 off-types, and 1 aberrant. The 3 normal plant weights are 101, 105, and 94. The 2 off-type plant weights are 84 and 88. The 1 aberrant plant weight is 32. Note in the results that for the group with only one response, the standard deviation is undefined and is set to NaN (not a number).

```csharp
using System;
using Imsl.Stat;
using Imsl.Math;

public class ANOVEx1
{
    public static void Main(String[] args)
    {
        double[][] y = { new double[]{101, 105, 94},
                        new double[]{84, 88},
                        new double[]{32}};
        ANOVA anova = new ANOVA(y);
        double[] aov = anova.GetArray();
        Console.Out.WriteLine("Degrees Of Freedom For Model = " + aov[0]);
        Console.Out.WriteLine("Degrees Of Freedom For Error = " + aov[1]);
        Console.Out.WriteLine("Total (Corrected) Degrees Of Freedom = " + aov[2]);
        Console.Out.WriteLine("Sum Of Squares For Model = " + aov[3]);
        Console.Out.WriteLine("Sum Of Squares For Error = " + aov[4]);
        Console.Out.WriteLine("Total (Corrected) Sum Of Squares = " + aov[5]);
        Console.Out.WriteLine("Model Mean Square = " + aov[6]);
        Console.Out.WriteLine("Error Mean Square = " + aov[7]);
        Console.Out.WriteLine("F statistic = " + aov[8]);
        Console.Out.WriteLine("P value= " + aov[9]);
        Console.Out.WriteLine("R Squared (in percent) = " + aov[10]);
        Console.Out.WriteLine("Adjusted R Squared (in percent) = " + aov[11]);
```
Model Error Standard deviation = " + aov[12]);
Console.Out.WriteLine("Mean Of Y = " + aov[13]);
Console.Out.WriteLine("Coefficient Of Variation (in percent) = " + aov[14]);
Console.Out.WriteLine("Total number of missing values = " + anova.TotalMissing);

PrintMatrixFormat pmf = new PrintMatrixFormat();
String[] labels =
    new String[] {"Group", "N", "Mean", "Std. Deviation"};
    pmf.SetColumnLabels(labels);
    pmf.NumberFormat = null;
    new PrintMatrix("Group Information").Print(pmf,
        (Object)anova.GetGroupInformation());

Output

Degrees Of Freedom For Model = 2
Degrees Of Freedom For Error = 3
Total (Corrected) Degrees Of Freedom = 5
Sum Of Squares For Model = 3480
Sum Of Squares For Error = 70
Total (Corrected) Sum Of Squares = 3550
Model Mean Square = 1740
Error Mean Square = 23.3333333333333
F statistic = 74.5714285714286
P value= 0.00276888252534979
R Squared (in percent) = 98.0281690140845
Adjusted R Squared (in percent) = 96.7136150234742
Model Error Standard deviation = 4.83045891539648
Mean Of Y = 84
Total number of missing values = 0

Group Information

<table>
<thead>
<tr>
<th>Group</th>
<th>N</th>
<th>Mean</th>
<th>Std. Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>5.56776436283002</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2.82842712474619</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>32</td>
<td>NaN</td>
</tr>
</tbody>
</table>

ANOVAFactorial Class

Analyzes a balanced factorial design with fixed effects.

public class Imsl.Stat.ANOVAFactorial
Properties

ErrorIncludeType

```java
public Imsl.Stat.ANOVAFactorial.ErrorCalculation ErrorIncludeType {get; set; }
```

The error included type. An int scalar specifying the included error type.

- **ANOVAFactorial.ErrorCalculation.Pure**, the default option, indicates factor `nSubscripts` is error. Its main effect and all its interaction effects are pooled into the error with the other `(ModelOrder + 1)`-way and higher-way interactions.
- **ANOVAFactorial.ErrorCalculation.Pooled** indicates factor `nSubscripts` is not error. Only `(ModelOrder + 1)`-way and higher-way interactions are included in the error.

ModelOrder

```java
public int ModelOrder {get; set; }
```

The number of factors to be included in the highest-way interaction in the model. An int scalar containing the number of factors to be included in the highest-way interaction in the model.

- `ModelOrder` must be in the interval `[1, nSubscripts-1]`. For example:
- `ModelOrder` of 1 indicates that a main effect model will be analyzed.
- `ModelOrder` of 2 indicates that two-way interactions will be included in the model.

Default: `ModelOrder = nSubscripts - 1`

Constructor

```
public ANOVAFactorial(int nSubscripts, int[] nLevels, double[] y)
```

Constructor for `ANOVAFactorial`.

- `y` must not contain NaN for any of its elements; i.e., missing values are not allowed.

- `nSubscripts` An int scalar containing the number of subscripts. Number of factors in the model + 1 (for the error term).

- `nLevels` An int array of length `nSubscripts` containing the number of levels for each of the factors for the first `nSubscripts-1` elements. `nLevels[nSubscripts-1]` is the number of observations per cell.

- `y` A double array of length `nLevels[0] * nLevels[1] * ... * nLevels[nSubscripts-1]` containing the responses.

`System.ArgumentException` is thrown if `nLevels.length`, and `y.length` are not consistent

Methods

`Compute`
public double Compute()
    Analyzes a balanced factorial design with fixed effects.
    Returns — A double scalar containing the p-value for the overall F test.

GetANOVATable
    public double[] GetANOVATable()
    Returns the analysis of variance table.

    The analysis of variance statistics are given as follows:

<table>
<thead>
<tr>
<th>Element</th>
<th>Analysis of Variance Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Degrees of freedom for the model</td>
</tr>
<tr>
<td>1</td>
<td>Degrees of freedom for error</td>
</tr>
<tr>
<td>2</td>
<td>Total (corrected) degrees of freedom</td>
</tr>
<tr>
<td>3</td>
<td>Sum of squares for the model</td>
</tr>
<tr>
<td>4</td>
<td>Sum of squares for error</td>
</tr>
<tr>
<td>5</td>
<td>Total (corrected) sum of squares</td>
</tr>
<tr>
<td>6</td>
<td>Model mean square</td>
</tr>
<tr>
<td>7</td>
<td>Error mean square</td>
</tr>
<tr>
<td>8</td>
<td>Overall $F$-statistic</td>
</tr>
<tr>
<td>9</td>
<td>$p$-value</td>
</tr>
<tr>
<td>10</td>
<td>$R^2$ (in percent)</td>
</tr>
<tr>
<td>11</td>
<td>Adjusted $R^2$ (in percent)</td>
</tr>
<tr>
<td>12</td>
<td>Estimate of the standard deviation</td>
</tr>
<tr>
<td>13</td>
<td>Overall mean of $y$</td>
</tr>
<tr>
<td>14</td>
<td>Coefficient of variation (in percent)</td>
</tr>
</tbody>
</table>

    Returns — A double array containing the analysis of variance table.

GetMeans
    public double[] GetMeans()
    Returns the subgroup means.
    Returns — A double array containing the subgroup means.

GetTestEffects
    public double[,] GetTestEffects()
    Returns statistics relating to the sums of squares for the effects in the model.
    Here,

    \[
    NEF = \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{\min(n, |\text{model order}|)}
    \]

    where $n$ is given by nSubscripts if ANOVAFactorial.ErrorCalculation.Pooled is specified; otherwise, nSubscripts-1. Suppose the factors are A, B, C, and error. With ModelOrder = 3, rows 0 through NEF-1 would correspond to A, B, C, AB, AC, BC, and ABC, respectively.

    The columns of the output matrix are as follows:
<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Degrees of freedom</td>
</tr>
<tr>
<td>1</td>
<td>Sum of squares</td>
</tr>
<tr>
<td>2</td>
<td>F-statistic</td>
</tr>
<tr>
<td>3</td>
<td>p-value</td>
</tr>
</tbody>
</table>

Returns — A double matrix containing statistics relating to the sums of squares for the effects in the model.

**Description**

Class ANOVAFactorial performs an analysis for an n-way classification design with balanced data. For balanced data, here must be an equal number of responses in each cell of the n-way layout. The effects are assumed to be fixed effects. The model is an extension of the two-way model to include n factors. The interactions (two-way, three-way, up to n-way) can be included in the model, or some of the higher-way interactions can be pooled into error. The ModelOrder property specifies the number of factors to be included in the highest-way interaction. For example, if three-way and higher-way interactions are to be pooled into error, set ModelOrder = 2. (By default, ModelOrder = nSubscripts - 1 with the last subscript being the error subscript.) Pure indicates there are repeated responses within the n-way cell; Pooled indicates otherwise.

Class ANOVAFactorial requires the responses as input into a single vector y in lexicographical order, so that the response subscript associated with the first factor varies least rapidly, followed by the subscript associated with the second factor, and so forth. Hemmerle (1967, Chapter 5) discusses the computational method.

**Example 1: Two-way Analysis of Variance**

A two-way analysis of variance is performed with balanced data discussed by Snedecor and Cochran (1967, Table 12.5.1, p. 347). The responses are the weight gains (in grams) of rats that were fed diets varying in the source (A) and level (B) of protein. The model is

\[ y_{ijk} = \mu + \alpha_i + \beta_j + \gamma_{ij} + \varepsilon_{ijk} \quad i = 1, 2; \ j = 1, 2, 3; \ k = 1, 2, \ldots, 10 \]

where

\[ \sum_{i=1}^{2} \alpha_i = 0; \ \sum_{j=1}^{3} \beta_j = 0; \ \sum_{i=1}^{2} \gamma_{ij} = 0 \quad \text{for } j = 1, 2, 3; \]

and

\[ \sum_{j=1}^{3} \gamma_{ij} = 0 \quad \text{for } j = 1, 2 \]
The first responses in each cell in the two-way layout are given in the following table:

<table>
<thead>
<tr>
<th>Protein Level (B)</th>
<th>Protein Source (A)</th>
<th>Beef</th>
<th>Cereal</th>
<th>Pork</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Beef</td>
<td>73, 102, 118, 104, 81, 107, 100, 87, 117, 111</td>
<td>98, 74, 56, 111, 95, 88, 82, 77, 86, 92</td>
<td>94, 79, 96, 98, 102, 102, 108, 91, 120, 105</td>
</tr>
<tr>
<td>Low</td>
<td>Beef</td>
<td>90, 76, 90, 64, 86, 51, 72, 90, 95, 78</td>
<td>107, 95, 97, 80, 98, 74, 74, 67, 89, 58</td>
<td>49, 82, 73, 86, 81, 97, 106, 70, 61, 82</td>
</tr>
</tbody>
</table>

```csharp
using System;
using Imsl.Stat;
using PrintMatrix = Imsl.Math.PrintMatrix;

class ANOVAfactorialEx1
{
    public static void Main(String[] args)
    {
        int nSubscripts = 3;
        int[] nLevels = new int[] {3, 2, 10};
        double[] y = new double[] {
            73.0, 102.0, 118.0, 104.0, 81.0, 107.0, 100.0, 87.0, 117.0, 111.0,
            90.0, 76.0, 90.0, 64.0, 86.0, 51.0, 72.0, 90.0, 95.0, 78.0,
            107.0, 95.0, 97.0, 80.0, 98.0, 74.0, 74.0, 67.0, 89.0, 58.0,
            49.0, 82.0, 73.0, 86.0, 81.0, 97.0, 106.0, 70.0, 61.0, 82.0
        };

        ANOVAfactorial af = new ANOVAfactorial(nSubscripts, nLevels, y);
        Console.Out.WriteLine (
            "P-value = " + af.Compute().ToString("0.000000"))
    }
}
```
**Output**

P-value = 0.002299

**Example 2: Two-way Analysis of Variance**

In this example, the same model and data is fit as in the example 1, but additional information is printed.

```csharp
using System;
using Imsl.Stat;

public class ANOVAFactorialEx2
{
    public static void Main(String[] args)
    {
        int nSubscripts = 3, i;
        int[] nLevels = new int[]{3, 2, 10};
        double[] y = new double[]{
            73.0, 102.0, 118.0,
            104.0, 81.0, 107.0,
            100.0, 87.0, 117.0,
            111.0, 90.0, 76.0,
            90.0, 64.0, 86.0,
            51.0, 72.0, 90.0,
            95.0, 78.0, 98.0,
            74.0, 56.0, 111.0,
            95.0, 88.0, 82.0,
            77.0, 86.0, 92.0,
            107.0, 95.0, 97.0,
            80.0, 98.0, 74.0,
            74.0, 67.0, 89.0,
            58.0, 94.0, 79.0,
            96.0, 98.0, 102.0,
            102.0, 108.0, 91.0,
            120.0, 105.0, 49.0,
            82.0, 73.0, 86.0,
            81.0, 97.0, 106.0,
            70.0, 61.0, 82.0};

        String[] labels =
            new String[]
                { "degrees of freedom for the model",
                  "degrees of freedom for error",
                  "total (corrected) degrees of freedom",
                  "sum of squares for the model",
                  "sum of squares for error",
                  "total (corrected) sum of squares",
                  "model mean square",
                  "error mean square",
                  "F-statistic",
                  "p-value",
                  "R-squared (in percent)",
                  "Adjusted R-squared (in percent)",
                  "est. standard deviation of the model error"};
```
ANOVAFactorial af =
   new ANOVAFactorial(nSubscripts, nLevels, y);

Console.Out.WriteLine
   ("P-value = " + af.Compute().ToString("0.000000");

Console.Out.WriteLine
   ("\n   * * * Analysis of Variance * * *");

for (i = 0; i < anova.Length; i++)
{  
   Console.Out.WriteLine
      (labels[i] + " " + anova[i].ToString("0.0000");
}

Console.Out.WriteLine
   ("\n   * * * Variation Due to the " + "Model * * *");

double[,] te = af.GetTestEffects();
for (i = 0; i < te.GetLength(0); i++)
{  
   Console.Out.WriteLine
      (labels[i] + " \t " + te[i,0].ToString("0.0000") + " \t " +
       te[i,1].ToString("0.0000") + " \t " +
       te[i,2].ToString("0.0000") + " \t " +
       te[i,3].ToString("0.0000");

Console.Out.WriteLine("\n   * * * Subgroup Means * * *");

double[] means = af.GetMeans();
for (i = 0; i < means.Length; i++)
{  
   Console.Out.WriteLine
      (mlabels[i] + " " + means[i].ToString("0.0000");
}
}

Output

P-value = 0.002299
**Analysis of Variance**

degrees of freedom for the model 5.0000
degrees of freedom for error 54.0000
total (corrected) degrees of freedom 59.0000
sum of squares for the model 4612.9333
sum of squares for error 11586.0000
total (corrected) sum of squares 16198.9333
model mean square 922.5867
error mean square 214.5556
F-statistic 4.3000
p-value 0.0023
R-squared (in percent) 28.4768
Adjusted R-squared (in percent) 21.8543
est. standard deviation of the model error 14.6477
overall mean of y 87.8667
coefficient of variation (in percent) 16.6704

**Variation Due to the Model**

Source DF Sum of Squares Mean Square Prob. of Larger F
A 2.0000 266.5333 0.6211 0.5411
B 1.0000 3168.2667 14.7666 0.0003
A*B 2.0000 1178.1333 2.7455 0.0732

**Subgroup Means**

grand mean 87.8667
A1 89.6000
A2 84.9000
A3 89.1000
B1 95.1333
B2 80.6000
A1*B1 100.0000
A2*B1 85.9000
A2*B2 83.9000
A3*B1 99.5000
A3*B2 78.7000

**Example 3: Three-way Analysis of Variance**

This example performs a three-way analysis of variance using data discussed by John (1971, pp. 91 92). The responses are weights (in grams) of roots of carrots grown with varying amounts of applied nitrogen (A), potassium (B), and phosphorus (C). Each cell of the three-way layout has one response. Note that the ABC interactions sum of squares, which is 186, is given incorrectly by John (1971, Table 5.2.) The three-way layout is given in the following table:

<table>
<thead>
<tr>
<th></th>
<th>$A_0$</th>
<th>$B_0$</th>
<th>$B_1$</th>
<th>$B_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_0$</td>
<td>88.76</td>
<td>91.41</td>
<td></td>
<td>97.85</td>
</tr>
<tr>
<td>$C_1$</td>
<td>87.45</td>
<td></td>
<td>98.27</td>
<td>95.83</td>
</tr>
<tr>
<td>$C_2$</td>
<td>86.01</td>
<td>104.20</td>
<td></td>
<td>90.09</td>
</tr>
</tbody>
</table>
using System;
using Imsl.Stat;

public class ANOVAFactorialEx3
{
    public static void Main(String[] args)
    {
        int nSubscripts = 3, i;
        int[] nLevels = new int[] {3, 3, 3};
        double[] y = new double[] {
            88.76, 87.45, 86.01, 91.41, 98.27, 104.2, 97.85, 95.85, 90.09,
            94.83, 84.57, 81.06, 100.49, 97.20, 120.8, 99.75, 112.3, 108.77,
            99.90, 92.98, 94.72, 100.23, 107.77, 118.39, 104.51, 110.94, 102.87,
        };
        String[] labels =
        new String[] {
            "degrees of freedom for the model",
            "degrees of freedom for error",
            "total (corrected) degrees of freedom",
            "sum of squares for the model",
            "sum of squares for error",
            "total (corrected) sum of squares",
            "model mean square",
            "error mean square",
            "F-statistic",
            "p-value",
            "R-squared (in percent)",
            "Adjusted R-squared (in percent)",
            "est. standard deviation of the model error",
            "overall mean of y",
            "coefficient of variation (in percent)"
        };
        String[] rlabels =
        new String[] {
        };
        ANOVAFactorial af =
        new ANOVAFactorial(nSubscripts, nLevels, y);
    }
}

### ANOVA Factorial Table

<table>
<thead>
<tr>
<th></th>
<th>A1</th>
<th></th>
<th>B1</th>
<th></th>
<th>B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>94.83</td>
<td>100.49</td>
<td>99.75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>84.57</td>
<td>97.20</td>
<td>112.30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>81.06</td>
<td>120.80</td>
<td>108.77</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A2</th>
<th></th>
<th>B1</th>
<th></th>
<th>B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>99.90</td>
<td>100.23</td>
<td>104.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>92.98</td>
<td>107.77</td>
<td>110.94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>94.72</td>
<td>118.39</td>
<td>102.87</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
af.ErrorIncludeType = ANOVAFactorial.ErrorCalculation.Pooled;
Console.Out.WriteLine
("P-value = " + af.Compute().ToString("0.000000"));

Console.Out.WriteLine
("\n  * * * Analysis of Variance * * *");
double[] anova = af.GetANOVATable();
for (i = 0; i < anova.Length; i++)
{
    Console.Out.WriteLine
        (labels[i] + " " + anova[i].ToString("0.0000"));
}

Console.Out.WriteLine
("\n  * * * Variation Due to the " + "Model * * *");
Console.Out.WriteLine
("Source\tDF\tSum of Squares\tMean Square\tProb. of Larger F"");
double[,] te = af.GetTestEffects();
for (i = 0; i < te.GetLength(0); i++)
{
    System.Text.StringBuilder sb =
        new System.Text.StringBuilder(rlabels[i]);

    int len = sb.Length;
    for (int j = 0; j < (8 - len); j++)
        sb.Append(' ');
    sb.Append(te[i,0].ToString("0.0000"));

    len = sb.Length;
    for (int j = 0; j < (16 - len); j++)
        sb.Append(' ');
    sb.Append(te[i,1].ToString("0.0000"));

    len = sb.Length;
    for (int j = 0; j < (32 - len); j++)
        sb.Append(' ');
    sb.Append(te[i,2].ToString("0.0000"));

    len = sb.Length;
    for (int j = 0; j < (48 - len); j++)
        sb.Append(' ');
    sb.Append(te[i,3].ToString("0.0000"));

    Console.Out.WriteLine(sb.ToString());
}
}

Output

P-value = 0.008299

Analysis of Variance  ANOVAFactorial Class • 355
### Analysis of Variance

<table>
<thead>
<tr>
<th>Source</th>
<th>DF</th>
<th>Sum of Squares</th>
<th>Mean Square</th>
<th>Prob. of Larger F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2.0000</td>
<td>488.3675</td>
<td>10.5152</td>
<td>0.0058</td>
</tr>
<tr>
<td>B</td>
<td>2.0000</td>
<td>1090.6564</td>
<td>23.4832</td>
<td>0.0004</td>
</tr>
<tr>
<td>C</td>
<td>2.0000</td>
<td>49.1485</td>
<td>1.0582</td>
<td>0.3911</td>
</tr>
<tr>
<td>A*B</td>
<td>4.0000</td>
<td>142.5853</td>
<td>1.5350</td>
<td>0.2804</td>
</tr>
<tr>
<td>A*C</td>
<td>4.0000</td>
<td>32.3474</td>
<td>0.3482</td>
<td>0.8383</td>
</tr>
<tr>
<td>B*C</td>
<td>4.0000</td>
<td>592.6238</td>
<td>6.3800</td>
<td>0.0131</td>
</tr>
</tbody>
</table>

### ANOVAFactorial.ErrorCalculation Enumeration

ErrorCalculation members indicate whether interaction effects are pooled into the error or not.

```csharp
public enumeration Imsl.Stat.ANOVAFactorial.ErrorCalculation

Fields

Pooled

public Imsl.Stat.ANOVAFactorial.ErrorCalculation Pooled
Indicates factor nSubscripts is not error.

Pure

public Imsl.Stat.ANOVAFactorial.ErrorCalculation Pure
Indicates factor nSubscripts is error. This is the default.
```
**MultipleComparisons Class**

Performs Student-Newman-Keuls multiple comparisons test.

```java
public class Imsl.Stat.MultipleComparisons
```

**Property**

**Alpha**

```java
public double Alpha {get; set; }
```

The significance level of the test. A `double` scalar containing the significance level of test.

**Alpha** must be in the interval `[0.01, 0.10]`. Default: `Alpha = 0.01`

**Constructor**

```java
public MultipleComparisons(double[] means, int df, double stdError)
```

Constructor for `MultipleComparisons`.

In fixed effects models, `stdError` equals the estimated standard error of a mean. For example, in a one-way model `stdError = √s^2/n` where `s^2` is the estimate of `σ^2` and `n` is the number of responses in a sample mean. In models with random components, use `stdError = sedif/√2` where `sedif` is the estimated standard error of the difference of two means.

- **means** A `double` array containing the means.
- **df** A `int` scalar containing the degrees of freedom associated with `stdError`.
- **stdError** A `double` scalar containing the effective estimated standard error of a mean.

**Method**

```java
public int[] Compute()
```

Performs Student-Newman-Keuls multiple comparisons test.

Value `equalMeans[I] = J` indicates the `I`-th smallest mean and the next `J-1` larger means are declared equal. Value `equalMeans[I] = 0` indicates no group of means starts with the `I`-th smallest mean.

Returns — A `int` array, call it `equalMeans`, indicating the size of the groups of means declared to be equal.
Description

Class MultipleComparisons performs a multiple comparison analysis of means using the Student-Newman-Keuls method. The null hypothesis is equality of all possible ordered subsets of a set of means. This null hypothesis is tested using the Studentized range of each of the corresponding subsets of sample means. The method is discussed in many elementary statistics texts, e.g., Kirk (1982, pp. 123-125).

Example: Multiple Comparisons Test

A multiple-comparisons analysis is performed using data discussed by Kirk (1982, pp. 123-125). The results show that there are three groups of means with three separate sets of values: (36.7, 40.3, 43.4), (40.3, 43.4, 47.2), and (43.4, 47.2, 48.7).

```csharp
using System;
using Imsl.Stat;
using PrintMatrix = Imsl.Math.PrintMatrix;

public class MultipleComparisonsEx1
{
    public static void Main(String[] args)
    {
        double[] means = new double[] {36.7, 48.7, 43.4, 47.2, 40.3};

        /* Perform multiple comparisons tests */
        MultipleComparisons mc =
            new MultipleComparisons(means, 45, 1.6970563);

        new PrintMatrix("Size of Groups of Means").Print(mc.Compute());
    }
}
```

Output

Size of Groups of Means

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
Chapter 15: Categorical and Discrete Data Analysis

Types

class ContingencyTable .............................................................. 359
class CategoricalGenLinModel ...................................................... 372
enumeration CategoricalGenLinModel.DistributionParameterModel ............... 390

Usage Notes

The ContingencyTable class computes many statistics of interest in a two-way table. Statistics computed by this routine include the usual chi-squared statistics, measures of association, Kappa, and many others.

ContingencyTable Class

Performs a chi-squared analysis of a two-way contingency table.

public class Imsl.Stat.ContingencyTable

Properties

ChiSquared

    public double ChiSquared {get; }
    Returns the Pearson chi-squared test statistic. A double scalar containing the Pearson chi-squared test statistic.

ContingencyCoef

359
public double ContingencyCoef {get; }
Returns contingency coefficient. A double scalar containing the contingency coefficient based on Pearson chi-squared statistic.

CramersV
public double CramersV {get; }
Returns Cramer’s V. A double scalar containing the Cramer’s V based on Pearson chi-squared statistic.

DegreesOfFreedom
public int DegreesOfFreedom {get; }
Returns the degrees of freedom for the chi-squared tests associated with the table. An int scalar containing the degrees of freedom for the chi-squared tests associated with the table.

ExactMean
public double ExactMean {get; }
Returns the exact mean. A double scalar containing the exact mean based on Pearson’s chi-square statistic.

ExactStdev
public double ExactStdev {get; }
Returns the exact standard deviation. A double scalar containing the exact standard deviation based on Pearson’s chi-square statistic.

GSquared
public double GSquared {get; }
Returns the likelihood ratio $G^2$ (chi-squared). A double scalar containing the likelihood ratio $G^2$ (chi-squared).

GSquaredP
public double GSquaredP {get; }
Returns the probability of a larger $G^2$ (chi-squared). A double scalar containing the probability of a larger $G^2$ (chi-squared).

P
public double P {get; }
Returns the Pearson chi-squared p-value for independence of rows and columns. A double scalar containing the Pearson chi-squared p-value for independence of rows and columns.

Phi
public double Phi {get; }
Returns phi. A double scalar containing the phi based on Pearson chi-squared statistic.

Constructor

ContingencyTable
public ContingencyTable(double[,] table)
Constructs and performs a chi-squared analysis of a two-way contingency table.

**table** A `double` matrix containing the observed counts in the contingency table.

## Methods

### GetContributions

```csharp
public double[,] GetContributions()
```
Returns the contributions to chi-squared for each cell in the table.

The last row and column contain the total contribution to chi-squared for that row or column.

**Returns** —
A `double` matrix of size `(table.GetLength(0)+1) * (table.GetLength(1)+1)` containing the contributions to chi-squared for each cell in the table.

### GetExpectedValues

```csharp
public double[,] GetExpectedValues()
```
Returns the expected values of each cell in the table.

The marginal totals are in the last row and column.

**Returns** —
A `double` matrix of size `(table.GetLength(0)+1) * (table.GetLength(1)+1)` containing the expected values of each cell in the table, under the null hypothesis.

### GetStatistics

```csharp
public double[,] GetStatistics()
```
Returns the statistics associated with this table.

Each row corresponds to a statistic.
The columns are as follows:

<table>
<thead>
<tr>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>estimated statistic</td>
</tr>
<tr>
<td>1</td>
<td>standard error for any parameter value</td>
</tr>
<tr>
<td>2</td>
<td>standard error under the null hypothesis</td>
</tr>
<tr>
<td>3</td>
<td>t value for testing the null hypothesis</td>
</tr>
<tr>
<td>4</td>
<td>p-value of the test in column 3</td>
</tr>
</tbody>
</table>

If a statistic cannot be computed, or if some value is not relevant for the computed statistic, the entry is NaN (Not a Number).

In the McNemar tests, column 0 contains the statistic, column 1 contains the chi-squared degrees of freedom, column 3 contains the exact p-value (1 degree of freedom only), and column 4 contains the chi-squared asymptotic p-value. The Kruskal-Wallis test is the same except no exact p-value is computed.

Returns — A double matrix of size 23 * 5 containing statistics associated with this table.
Description

Class ContingencyTable computes statistics associated with an \( r \times c \) contingency table. The function computes the chi-squared test of independence, expected values, contributions to chi-squared, row and column marginal totals, some measures of association, correlation, prediction, uncertainty, the McNemar test for symmetry, a test for linear trend, the odds and the log odds ratio, and the kappa statistic (if the appropriate optional arguments are selected).

Notation

Let \( x_{ij} \) denote the observed cell frequency in the \( ij \) cell of the table and \( n \) denote the total count in the table. Let \( p_{ij} = p_i \cdot p_j \cdot \) denote the predicted cell probabilities under the null hypothesis of independence, where \( p_i \cdot \) and \( p_j \cdot \) are the row and column marginal relative frequencies. Next, compute the expected cell counts as \( e_{ij} = np_{ij} \).

Also required in the following are \( a_{uv} \) and \( b_{uv} \) for \( u, v = 1, \ldots, n \). Let \( (r_s, c_s) \) denote the row and column response of observation \( s \). Then, \( a_{uv} = 1, 0, \) or \(-1\), depending on whether \( r_u < r_v, r_u = r_v, \) or \( r_u > r_v \), respectively. The \( b_{uv} \) are similarly defined in terms of the \( c_s \) variables.

Chi-squared Statistic

For each cell in the table, the contribution to \( \chi^2 \) is given as \( (x_{ij} - e_{ij})^2 / e_{ij} \). The Pearson chi-squared statistic (denoted \( \chi^2 \)) is computed as the sum of the cell contributions to chi-squared. It has \((r - 1)(c - 1)\) degrees of freedom and tests the null hypothesis of independence, i.e., \( H_0 : p_{ij} = p_i \cdot p_j \cdot \). The null hypothesis is rejected if the computed value of \( \chi^2 \) is too large.

The maximum likelihood equivalent of \( \chi^2, G^2 \) is computed as follows:

\[
G^2 = -2 \sum_{i,j} x_{ij} \ln \left( \frac{x_{ij}}{n p_{ij}} \right)
\]

\( G^2 \) is asymptotically equivalent to \( \chi^2 \) and tests the same hypothesis with the same degrees of freedom.

Measures Related to Chi-squared (Phi, Contingency Coefficient, and Cramer’s V)

There are three measures related to chi-squared that do not depend on sample size:

\[
\text{phi, } \phi = \sqrt{\frac{\chi^2}{n}}
\]

contingency coefficient, \( P = \sqrt{\frac{\chi^2}{(n + \chi^2)}} \)

Cramer’s \( V, V = \sqrt{\frac{\chi^2}{(n \min (r, c))}} \)
Since these statistics do not depend on sample size and are large when the hypothesis of independence is rejected, they can be thought of as measures of association and can be compared across tables with different sized samples. While both $P$ and $V$ have a range between 0.0 and 1.0, the upper bound of $P$ is actually somewhat less than 1.0 for any given table (see Kendall and Stuart 1979, p. 587). The significance of all three statistics is the same as that of the $\chi^2$ statistic, which is contained in the `ChiSquared` property.

The distribution of the $\chi^2$ statistic in finite samples approximates a chi-squared distribution. To compute the exact mean and standard deviation of the $\chi^2$ statistic, Haldane (1939) uses the multinomial distribution with fixed table marginals. The exact mean and standard deviation generally differ little from the mean and standard deviation of the associated chi-squared distribution.

### Standard Errors and p-values for Some Measures of Association

In Columns 1 through 4 of statistics, estimated standard errors and asymptotic p-values are reported. Estimates of the standard errors are computed in two ways. The first estimate, in Column 1 of the return matrix from the `Statistics` property, is asymptotically valid for any value of the statistic. The second estimate, in Column 2 of the array, is only correct under the null hypothesis of no association. The $z$-scores in Column 3 of statistics are computed using this second estimate of the standard errors. The $p$-values in Column 4 are computed from this $z$-score. See Brown and Benedetti (1977) for a discussion and formulas for the standard errors in Column 2.

### Measures of Association for Ranked Rows and Columns

The measures of association, $\phi$, $P$, and $V$, do not require any ordering of the row and column categories. Class `ContingencyTable` also computes several measures of association for tables in which the rows and column categories correspond to ranked observations. Two of these tests, the product-moment correlation and the Spearman correlation, are correlation coefficients computed using assigned scores for the row and column categories. The cell indices are used for the product-moment correlation, while the average of the tied ranks of the row and column marginals is used for the Spearman rank correlation. Other scores are possible.

Gamma, Kendall’s $\tau_b$, Stuart’s $\tau_c$, and Somers’ $D$ are measures of association that are computed like a correlation coefficient in the numerator. In all these measures, the numerator is computed as the ”covariance” between the $a_{uv}$ variables and $b_{uv}$ variables defined above, i.e., as follows:

$$\sum_u \sum_v a_{uv}b_{uv}$$

Recall that $a_{uv}$ and $b_{uv}$ can take values -1, 0, or 1. Since the product $a_{uv}b_{uv} = 1$ only if $a_{uv}$ and $b_{uv}$ are both 1 or are both -1, it is easy to show that this ”covariance” is twice the total number of agreements minus the number of disagreements, where a disagreement occurs when $a_{uv}b_{uv} = -1$.

Kendall’s $\tau_b$ is computed as the correlation between the $a_{uv}$ variables and the $b_{uv}$ variables (see Kendall and Stuart 1979, p. 593). In a rectangular table ($r \neq c$), Kendall’s $\tau_b$ cannot be 1.0 (if
all marginal totals are positive). For this reason, Stuart suggested a modification to the denominator of \( \tau \) in which the denominator becomes the largest possible value of the "covariance." This maximizing value is approximately \( n^2 m / (m - 1) \), where \( m = \min (r, c) \). Stuart’s \( \tau_c \) uses this approximate value in its denominator. For large \( n \), \( \tau_c \approx m \tau_b / (m - 1) \).

Gamma can be motivated in a slightly different manner. Because the "covariance" of the \( a_{uv} \) variables and the \( b_{uv} \) variables can be thought of as twice the number of agreements minus the disagreements, \( 2 (A - D) \), where \( A \) is the number of agreements and \( D \) is the number of disagreements, Gamma is motivated as the probability of agreement minus the probability of disagreement, given that either agreement or disagreement occurred. This is shown as \( \gamma = (A - D) / (A + D) \).

Two definitions of Somers’ \( D \) are possible, one for rows and a second for columns. Somers’ \( D \) for rows can be thought of as the regression coefficient for predicting \( a_{uv} \) from \( b_{uv} \). Moreover, Somer’s \( D \) for rows is the probability of agreement minus the probability of disagreement, given that the column variable, \( b_{uv} \), is not 0. Somers’ \( D \) for columns is defined in a similar manner.

A discussion of all of the measures of association in this section can be found in Kendall and Stuart (1979, p. 592).

**Measures of Prediction and Uncertainty**

**Optimal Prediction Coefficients:** The measures in this section do not require any ordering of the row or column variables. They are based entirely upon probabilities. Most are discussed in Bishop et al. (1975, p. 385).

Consider predicting (or classifying) the column for a given row in the table. Under the null hypothesis of independence, choose the column with the highest column marginal probability for all rows. In this case, the probability of misclassification for any row is 1 minus this marginal probability. If independence is not assumed within each row, choose the column with the highest row conditional probability. The probability of misclassification for the row becomes 1 minus this conditional probability.

Define the optimal prediction coefficient \( \lambda_{c|r} \) for predicting columns from rows as the proportion of the probability of misclassification that is eliminated because the random variables are not independent. It is estimated by

\[
\lambda_{c|r} = \frac{(1 - p_{•m}) - (1 - \sum_i p_{im})}{1 - p_{•m}}
\]

where \( m \) is the index of the maximum estimated probability in the row \( (p_{im}) \) or row margin \( (p_{•m}) \). A similar coefficient is defined for predicting the rows from the columns. The symmetric version of the optimal prediction \( \lambda \) is obtained by summing the numerators and denominators of \( \lambda_{c|r} \) and \( \lambda_{r|c} \) then dividing. Standard errors for these coefficients are given in Bishop et al. (1975, p. 388).

A problem with the optimal prediction coefficients \( \lambda \) is that they vary with the marginal probabilities. One way to correct this is to use row conditional probabilities. The optimal prediction \( \lambda^* \) coefficients are defined as the corresponding \( \lambda \) coefficients in which first the row
(or column) marginals are adjusted to the same number of observations. This yields

$$\lambda^*_{c|r} = \frac{\sum_i \max_j p_{j|i} - \max_j (\sum_i p_{j|i})}{R - \max_j (\sum_i p_{j|i})}$$

where $i$ indexes the rows, $j$ indexes the columns, and $p_{j|i}$ is the (estimated) probability of column $j$ given row $i$.

$$\lambda^*_{r|c}$$

is similarly defined.

**Goodman and Kruskal $\tau$:** A second kind of prediction measure attempts to explain the proportion of the explained variation of the row (column) measure given the column (row) measure. Define the total variation in the rows as follows:

$$n/2 - \left(\sum_i x^2_{i\cdot}\right)/(2n)$$

Note that this is $1/(2n)$ times the sums of squares of the $a_{uv}$ variables.

With this definition of variation, the Goodman and Kruskal $\tau$ coefficient for rows is computed as the reduction of the total variation for rows accounted for by the columns, divided by the total variation for the rows. To compute the reduction in the total variation of the rows accounted for by the columns, note that the total variation for the rows within column $j$ is defined as follows:

$$q_j = x_{\cdot j}/2 - \left(\sum_i x^2_{ij}\right)/(2x_{\cdot j})$$

The total variation for rows within columns is the sum of the $q_j$ variables. Consistent with the usual methods in the analysis of variance, the reduction in the total variation is given as the difference between the total variation for rows and the total variation for rows within the columns.

Goodman and Kruskal’s $\tau$ for columns is similarly defined. See Bishop et al. (1975, p. 391) for the standard errors.

**Uncertainty Coefficients:** The uncertainty coefficient for rows is the increase in the log-likelihood that is achieved by the most general model over the independence model, divided by the marginal log-likelihood for the rows. This is given by the following equation:
The uncertainty coefficient for columns is similarly defined. The symmetric uncertainty coefficient contains the same numerator as $U_{r|c}$ and $U_{c|r}$ but averages the denominators of these two statistics. Standard errors for $U$ are given in Brown (1983).

**Kruskal-Wallis:** The Kruskal-Wallis statistic for rows is a one-way analysis-of-variance-type test that assumes the column variable is monotonically ordered. It tests the null hypothesis that no row populations are identical, using average ranks for the column variable. The Kruskal-Wallis statistic for columns is similarly defined. Conover (1980) discusses the Kruskal-Wallis test.

**Test for Linear Trend:** When there are two rows, it is possible to test for a linear trend in the row probabilities if it is assumed that the column variable is monotonically ordered. In this test, the probabilities for row 1 are predicted by the column index using weighted simple linear regression. This slope is given by

$$
\hat{\beta} = \frac{\sum_{j} x_{\cdot j} (x_{1j}/x_{\cdot j} - x_{1\cdot}/n) (j - \bar{j})}{\sum_{j} x_{\cdot j} (j - \bar{j})^2}
$$

where

$$
\bar{j} = \sum_{j} x_{\cdot j} j/n
$$

is the average column index. An asymptotic test that the slope is 0 may then be obtained (in large samples) as the usual regression test of zero slope.

In two-column data, a similar test for a linear trend in the column probabilities is computed. This test assumes that the rows are monotonically ordered.

**Kappa:** Kappa is a measure of agreement computed on square tables only. In the kappa statistic, the rows and columns correspond to the responses of two judges. The judges agree along the diagonal and disagree off the diagonal. Let

$$
p_0 = \sum_{i} x_{ii}/n
$$

denote the probability that the two judges agree, and let

$$
p_c = \sum_{i} e_{ii}/n
$$
denote the expected probability of agreement under the independence model. Kappa is then given by \((p_0 - p_c)/(1 - p_c)\).

**McNemar Tests:** The McNemar test is a test of symmetry in a square contingency table. In other words, it is a test of the null hypothesis \(H_0 : \theta_{ij} = \theta_{ji}\). The multiple degrees-of-freedom version of the McNemar test with \(r (r - 1)/2\) degrees of freedom is computed as follows:

\[
\sum_{i<j} \frac{(x_{ij} - x_{ji})^2}{(x_{ij} + x_{ji})}
\]

The single degree-of-freedom test assumes that the differences, \(x_{ij} - x_{ji}\), are all in one direction. The single degree-of-freedom test will be more powerful than the multiple degrees-of-freedom test when this is the case. The test statistic is given as follows:

\[
\frac{\left(\sum_{i<j} (x_{ij} - x_{ji})\right)^2}{\sum_{i<j} (x_{ij} + x_{ji})}
\]

The exact probability can be computed by the binomial distribution.

**Example 1: Contingency Table**

The following example is taken from Kendall and Stuart (1979) and involves the distance vision in the right and left eyes.

```csharp
using System;
using Imsl.Stat;

class ContingencyTableEx1
{
    static void Main(String[] args)
    {
        double[,] table = {{821, 112, 85, 35},
                           {116, 494, 145, 27},
                           {72, 151, 583, 87},
                           {43, 34, 106, 331}};

        ContingencyTable ct = new ContingencyTable(table);
        Console.Out.WriteLine("P-value = " + ct.P);
    }
}
```

**ContingencyTable Class IMSL C# Numerical Library**
Output

P-value = 0

Example 2: Contingency Table

The following example, which illustrates the use of Kappa and McNemar tests, uses the same distance vision data as in Example 1.

```csharp
using System;
using Imsl.Stat;
using Imsl.Math;

public class ContingencyTableEx2
{
    public static void Main(String[] args)
    {
        double[,] table = {{821.0, 112.0, 85.0, 35.0},
                            {116.0, 494.0, 145.0, 27.0},
                            {72.0, 151.0, 583.0, 87.0},
                            {43.0, 34.0, 106.0, 331.0}};
        String[] rlabels = new String[] {"Gamma", "Tau B",
                                           "Tau C", "D-Row",
                                           "D-Column", "Correlation",
                                           "Spearman", "GK tau rows",
                                           "GK tau cols.", "U - sym."
                                           "U - rows", "U - cols."
                                           "Lambda-sym.", "Lambda-row"
                                           "Lambda-col."
                                           "1-star-rows"
                                           "1-star-col."
                                           "Lin. trend"
                                           "Kruskal row"
                                           "Kruskal col.", "Kappa"
                                           "McNemar"
                                           "McNemar df=1"};
        ContingencyTable ct = new ContingencyTable(table);

        Console.Out.WriteLine("Pearson chi-squared statistic = " +
                              ct.ChiSquared.ToString("0.0000");
        Console.Out.WriteLine("p-value for Pearson chi-squared = " +
                              ct.P.ToString("0.0000");
        Console.Out.WriteLine("degrees of freedom = " +
                              ct.DegreesOfFreedom);
        Console.Out.WriteLine("G-squared statistic = " +
                              ct.GSquared.ToString("0.0000");
        Console.Out.WriteLine("p-value for G-squared = " +
                              ct.GSquaredP.ToString("0.0000");
        Console.Out.WriteLine("degrees of freedom = " +
                              ct.DegreesOfFreedom);

        PrintMatrix pm = new PrintMatrix("\n* * * Table Values * * *");
```
PrintMatrixFormat pmf = new PrintMatrixFormat();

pmf.NumberFormat = "0.00";

pm.Print(pmf, table);

pmf.NumberFormat = "0.0000";

pmf.SetTitle("* * * Contributions to Chi-squared* * *");

pmf.Print(pmf, ct.GetContributions());

pmf.NumberFormat = "0.0000";

pmf.SetTitle("* * * Contributions to Chi-squared* * *");

pm.Print(pmf, ct.GetExpectedValues());

Console.Out.WriteLine("* * * Expected Values * * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

Console.Out.WriteLine("* * * Contributions to Chi-squared* * *");

System.Text.StringBuilder sb =
new System.Text.StringBuilder(rlabels[i]);

int len = sb.Length;
for (int j = 0; j < (13 - len); j++)
    sb.Append(' ');

sb.Append(stat[i,0].ToString("0.0000"));

len = sb.Length;
for (int j = 0; j < (24 - len); j++)
    sb.Append(' ');

sb.Append(stat[i,1].ToString("0.0000"));

len = sb.Length;
for (int j = 0; j < (36 - len); j++)
    sb.Append(' ');

sb.Append(stat[i,2].ToString("0.0000"));

len = sb.Length;
for (int j = 0; j < (50 - len); j++)
    sb.Append(' ');

sb.Append(stat[i,3].ToString("0.0000"));

len = sb.Length;
for (int j = 0; j < (63 - len); j++)
    sb.Append(' ');

sb.Append(stat[i,4].ToString("0.0000"));

Console.Out.WriteLine(sb.ToString());

370 • ContingencyTable Class | IMSL C# Numerical Library
Output

Pearson chi-squared statistic = 3304.3684
p-value for Pearson chi-squared = 0.0000
degrees of freedom = 9
G-squared statistic = 2781.0190
p-value for G-squared = 0.0000
degrees of freedom = 9

* * * Table Values * * *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>821.00</td>
<td>112.00</td>
<td>85.00</td>
<td>35.00</td>
</tr>
<tr>
<td>1</td>
<td>116.00</td>
<td>494.00</td>
<td>145.00</td>
<td>27.00</td>
</tr>
<tr>
<td>2</td>
<td>72.00</td>
<td>151.00</td>
<td>583.00</td>
<td>87.00</td>
</tr>
<tr>
<td>3</td>
<td>43.00</td>
<td>34.00</td>
<td>106.00</td>
<td>331.00</td>
</tr>
</tbody>
</table>

* * * Expected Values * * *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>341.69</td>
<td>256.92</td>
<td>298.49</td>
<td>155.90</td>
<td>1053.00</td>
</tr>
<tr>
<td>1</td>
<td>253.75</td>
<td>190.80</td>
<td>221.67</td>
<td>115.78</td>
<td>782.00</td>
</tr>
<tr>
<td>2</td>
<td>166.79</td>
<td>125.41</td>
<td>145.70</td>
<td>76.10</td>
<td>514.00</td>
</tr>
<tr>
<td>3</td>
<td>1052.00</td>
<td>791.00</td>
<td>919.00</td>
<td>480.00</td>
<td>3242.00</td>
</tr>
</tbody>
</table>

* * * Contributions to Chi-squared* * *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>672.3626</td>
<td>81.7416</td>
<td>152.6959</td>
<td>93.7612</td>
<td>1000.5613</td>
</tr>
<tr>
<td>1</td>
<td>74.7802</td>
<td>481.8351</td>
<td>26.5189</td>
<td>68.0768</td>
<td>651.2109</td>
</tr>
<tr>
<td>2</td>
<td>163.6605</td>
<td>20.5287</td>
<td>429.8489</td>
<td>15.4625</td>
<td>629.5006</td>
</tr>
<tr>
<td>3</td>
<td>91.8743</td>
<td>66.6263</td>
<td>10.8183</td>
<td>853.7768</td>
<td>1023.0957</td>
</tr>
<tr>
<td>4</td>
<td>1002.6776</td>
<td>650.7317</td>
<td>619.8819</td>
<td>1031.0772</td>
<td>3304.3684</td>
</tr>
</tbody>
</table>

* * * Chi-square Statistics * * *

Exact mean = 9.0028
Exact standard deviation = 4.2402
Phi = 1.0096
P = 0.7105
Cramer’s V = 0.5829

<table>
<thead>
<tr>
<th></th>
<th>stat.</th>
<th>std. err.</th>
<th>std. err.(Ho)</th>
<th>t-value(Ho)</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gamma</td>
<td>0.7757</td>
<td>0.0123</td>
<td>0.0149</td>
<td>52.1897</td>
<td>0.0000</td>
</tr>
<tr>
<td>Tau B</td>
<td>0.6429</td>
<td>0.0122</td>
<td>0.0123</td>
<td>52.1897</td>
<td>0.0000</td>
</tr>
<tr>
<td>Tau C</td>
<td>0.6293</td>
<td>0.0121</td>
<td>NaN</td>
<td>52.1897</td>
<td>0.0000</td>
</tr>
<tr>
<td>D-Row</td>
<td>0.6418</td>
<td>0.0122</td>
<td>0.0123</td>
<td>52.1897</td>
<td>0.0000</td>
</tr>
<tr>
<td>D-Column</td>
<td>0.6439</td>
<td>0.0122</td>
<td>0.0123</td>
<td>52.1897</td>
<td>0.0000</td>
</tr>
<tr>
<td>Correlation</td>
<td>0.6926</td>
<td>0.0128</td>
<td>0.0172</td>
<td>40.2669</td>
<td>0.0000</td>
</tr>
<tr>
<td>Spearman</td>
<td>0.6939</td>
<td>0.0127</td>
<td>0.0127</td>
<td>54.6614</td>
<td>0.0000</td>
</tr>
<tr>
<td>GK tau rows</td>
<td>0.3420</td>
<td>0.0123</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>GK tau cols.</td>
<td>0.3430</td>
<td>0.0122</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
### CategoricalGenLinModel Class

Analyzes categorical data using logistic, probit, Poisson, and other linear models.

```csharp
public class Imsl.Stat.CategoricalGenLinModel

<table>
<thead>
<tr>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaseAnalysis</td>
</tr>
</tbody>
</table>

#### CaseAnalysis

```csharp
virtual public double[,] CaseAnalysis {get;}
```

The case analysis. A double matrix containing the case analysis or null if `Imsl.Stat.CategoricalGenLinModel.Solve` has not been called. The matrix is `nobs × 5` where `nobs` is the number of observations. The matrix contains:

<table>
<thead>
<tr>
<th>Column</th>
<th>Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Prediction.</td>
</tr>
<tr>
<td>1</td>
<td>The residual.</td>
</tr>
<tr>
<td>2</td>
<td>The estimated standard error of the residual.</td>
</tr>
<tr>
<td>3</td>
<td>The estimated influence of the observation.</td>
</tr>
<tr>
<td>4</td>
<td>The standardized residual.</td>
</tr>
</tbody>
</table>

Case studies are computed for all observations except where missing values prevent their computation. The prediction in column 0 depends upon the model used as follows:

<table>
<thead>
<tr>
<th>Model</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The predicted mean for the observation.</td>
</tr>
<tr>
<td>1-4</td>
<td>The probability of a success on a single trial.</td>
</tr>
</tbody>
</table>
CensorColumn

```csharp
virtual public int CensorColumn { set; }
```

The column number in \( x \) which contains the interval type for each observation. An \( \text{int} \) scalar which indicates the column number \( x \) which contains the interval type code for each observation.

The valid codes are interpreted as:

<table>
<thead>
<tr>
<th>( x[i,\text{CensorColumn}] )</th>
<th>Censoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Point observation. The response is unique and is given by ( x[i,\text{LowerEndpointColumn}] ).</td>
</tr>
<tr>
<td>1</td>
<td>Right interval. The response is greater than or equal to ( x[i,\text{LowerEndpointColumn}] ) and less than or equal to the upper bound, if any, of the distribution.</td>
</tr>
<tr>
<td>2</td>
<td>Left interval. The response is less than or equal to ( x[i,\text{UpperEndpointColumn}] ) and greater than or equal to the lower bound of the distribution.</td>
</tr>
<tr>
<td>3</td>
<td>Full interval. The response is greater than or equal to ( x[i,\text{LowerEndpointColumn}] ) but less than or equal to ( x[i,\text{UpperEndpointColumn}] ).</td>
</tr>
</tbody>
</table>

Default: \( \text{CensorColumn} = 0 \).

ClassificationVariableColumn

```csharp
virtual public int[] ClassificationVariableColumn { set; }
```

An index vector to contain the column numbers in \( x \) that are classification variables. An \( \text{int} \) vector which contains the column numbers in \( x \) that are classification variables.

By default this vector is not referenced.

ClassificationVariableCounts

```csharp
virtual public int[] ClassificationVariableCounts { get; }
```

The number of values taken by each classification variable. An \( \text{int} \) array of length \( n_{clvar} \) containing the number of values taken by each classification variable where \( n_{clvar} \) is the number of classification variables or \( \text{null} \) if \( \text{Imsl.Stat.CategoricalGenLinModel.Solve} \) (p. 380) has not been called.

ClassificationVariableValues

```csharp
virtual public double[] ClassificationVariableValues { get; }
```

The distinct values of the classification variables in ascending order. A \( \text{double} \) array of length \( \sum_{k=0}^{n_{clvar}} n_{clval}[k] \) containing the distinct values of the classification variables in ascending order where \( n_{clvar} \) is the number of classification variables and \( n_{clval}[i] \) is the number of values taken by the \( i \)-th classification variable.

A \( \text{null} \) is returned if \( \text{Imsl.Stat.CategoricalGenLinModel.Solve} \) (p. 380) has not been called prior to calling this method.

ConvergenceTolerance

```csharp
virtual public double ConvergenceTolerance { set; }
```
The convergence criterion. A double scalar specifying the convergence criterion.

Convergence is assumed when the maximum relative change in any coefficient estimate is less than ConvergenceTolerance from one iteration to the next or when the relative change in the log-likelihood, Imsl.Stat.CategoricalGenLinModel.OptimizedCriterion (p. 377), from one iteration to the next is less than ConvergenceTolerance/100. ConvergenceTolerance must be greater than 0.

Default: ConvergenceTolerance = .001.

CovarianceMatrix

virtual public double[,] CovarianceMatrix {get; }

The estimated asymptotic covariance matrix of the coefficients. A double matrix containing the estimated asymptotic covariance matrix of the coefficients or null if Imsl.Stat.CategoricalGenLinModel.Solve (p. 380) has not been called.

The covariance matrix is nCoef by nCoef where nCoef is the number of coefficients in the model.

DesignVariableMeans

virtual public double[] DesignVariableMeans {get; }

The means of the design variables. A double array of length nCoef containing the means of the design variables where nCoef is the number of coefficients in the model or null if Imsl.Stat.CategoricalGenLinModel.Solve (p. 380) has not been called.

ExtendedLikelihoodObservations

virtual public int[] ExtendedLikelihoodObservations {get; set; }

A vector indicating which observations are included in the extended likelihood. An int array of length nobs indicating which observations are included in the extended likelihood where nobs is the number of observations.

ExtendedLikelihoodObservations is an int array of length nobs indicating which observations are included in the extended likelihood where nobs is the number of observations. The values within the array are interpreted as:

<table>
<thead>
<tr>
<th>Value</th>
<th>Status of observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Observation i is in the likelihood.</td>
</tr>
<tr>
<td>1</td>
<td>Observation i cannot be in the likelihood because it contains at least one missing value in x.</td>
</tr>
<tr>
<td>2</td>
<td>Observation i is not in the likelihood. Its estimated parameter is infinite.</td>
</tr>
</tbody>
</table>

A null is returned if Imsl.Stat.CategoricalGenLinModel.Solve (p. 380) has not been called prior to calling this method.

Default: All elements are zero.

FixedParameterColumn

virtual public int FixedParameterColumn {set; }

The column number in x that contains a fixed parameter for each observation that is added to the linear response prior to computing the model parameter. An int scalar
which indicates the column number in $x$ that contains a fixed parameter for each observation that is added to the linear response prior to computing the model parameter.

The "fixed" parameter allows one to test hypothesis about the parameters via the log-likelihoods. By default the fixed parameter is assumed to be zero.

**FrequencyColumn**

```csharp
virtual public int FrequencyColumn {set; }
```

The column number in $x$ that contains the frequency of response for each observation. An int scalar which indicates the column number in $x$ that contains the frequency of response for each observation.

By default a frequency of 1 for each observation is assumed.

**Hessian**

```csharp
virtual public double[,] Hessian {get; }
```

The Hessian computed at the initial parameter estimates. A double matrix containing the Hessian computed at the input parameter estimates.

The Hessian matrix is $nCoef$ by $nCoef$ where $nCoef$ is the number of coefficients in the model. This member function will call Imsl.Stat.CategoricalGenLinModel.Solve (p. 380) to get the Hessian if the Hessian has not already been computed.

**InfiniteEstimateMethod**

```csharp
virtual public int InfiniteEstimateMethod {set; }
```

Specifies the method used for handling infinite estimates. An int scalar indicating which method to use for handling infinite estimates.

The value of **InfiniteEstimateMethod** is interpreted as follows:

<table>
<thead>
<tr>
<th>InfiniteEstimateMethod</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Remove a right or left-censored observation from the log-likelihood whenever the probability of the observation exceeds 0.995. At convergence, use linear programming to check that all removed observations actually have an estimated linear response that is infinite. Set <code>ExtendedLikelihoodObservations[i]</code> for observation $i$ to 2 if the linear response is infinite. If not all removed observations have infinite linear response, recompute the estimates based upon the observations with estimated linear response that is finite. This option is valid only for censoring codes 1 and 2.</td>
</tr>
<tr>
<td>1</td>
<td>Iterate without checking for infinite estimates.</td>
</tr>
</tbody>
</table>

By default `InfiniteEstimateMethod = 1`.

**LastParameterUpdates**

```csharp
virtual public double[] LastParameterUpdates {get; }
```
The last parameter updates (excluding step halvings). A `double` array of length `nCoef` containing the last parameter updates (excluding step halvings) or `null` if `Imsl.Stat.CategoricalGenLinModel.Solve` (p. 380) has not been called.

**LowerEndpointColumn**

```csharp
virtual public int LowerEndpointColumn {set; }
```

The column number in `x` that contains the lower endpoint of the observation interval for full interval and right interval observations. An `int` scalar which indicates the column number in `x` that contains the lower endpoint of the observation interval for full interval and right interval observations.

By default all observations are treated as "point" observations and `x[i,LowerEndpointColumn]` contains the observation point. If this member function is not called, the last column of `x` is assumed to contain the "point" observations.

**MaxIterations**

```csharp
virtual public int MaxIterations {set; }
```

The maximum number of iterations allowed. An `int` specifying the maximum number of iterations allowed.

Default: `MaxIterations = 30`.

**ModelIntercept**

```csharp
virtual public int ModelIntercept {set; }
```

The intercept option. An `int` scalar which indicates whether or not the model has an intercept.

Input `ModelIntercept` is interpreted as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No intercept is in the model (unless otherwise provided for by the user).</td>
</tr>
<tr>
<td>1</td>
<td>Intercept is automatically included in the model.</td>
</tr>
</tbody>
</table>

By default `ModelIntercept = 1`.

**NRowsMissing**

```csharp
virtual public int NRowsMissing {get; }
```

The number of rows of data in `x` that contain missing values in one or more specific columns of `x`. An `int` scalar representing the number of rows of data in `x` that contain missing values in one or more specific columns of `x` or `null` if `Imsl.Stat.CategoricalGenLinModel.Solve` (p. 380) has not been called.

The columns of `x` included in the count are the columns containing the upper or lower endpoints of full interval, left interval, or right interval observations. Also included are the columns containing the frequency responses, fixed parameters, optional distribution parameters, and interval type for each observation. Columns containing classification variables and columns associated with each effect in the model are also included.

**ObservationMax**

```csharp
virtual public int ObservationMax {set; }
```
The maximum number of observations that can be handled in the linear programming. An int scalar which sets the maximum number of observations that can be handled in the linear programming.

Default: ObservationMax is set to the number of observations.

OptimizedCriterion

```csharp
virtual public double OptimizedCriterion { get; }
```

The optimized criterion. A double scalar representing the optimized criterion or null if Imsl.Stat.CategoricalGenLinModel.Solve (p. 380) has not been called.

The criterion to be maximized is a constant plus the log-likelihood.

OptionalDistributionParameterColumn

```csharp
virtual public int OptionalDistributionParameterColumn { set; }
```

The column number in x that contains an optional distribution parameter for each observation. An int scalar which indicates the column number in x that contains an optional distribution parameter for each observation.

The distribution parameter values are interpreted as follows depending on the model chosen:

<table>
<thead>
<tr>
<th>Model</th>
<th>Meaning of x[i,OptionalDistributionParameterColumn]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The Poisson parameter is given by x[i,OptionalDistributionParameterColumn] × e^ρ.</td>
</tr>
<tr>
<td>1</td>
<td>The number of successes required in the negative binomial is given by x[i,OptionalDistributionParameterColumn].</td>
</tr>
<tr>
<td>2</td>
<td>x[i,OptionalDistributionParameterColumn] is not used.</td>
</tr>
<tr>
<td>3-5</td>
<td>The number of trials in the binomial distribution is given by x[i,OptionalDistributionParameterColumn].</td>
</tr>
</tbody>
</table>

By default the distribution parameter is assumed to be 1.

Parameters

```csharp
virtual public double[,] Parameters { get; }
```

Parameter estimates and associated statistics. An nCoef row by 4 column double matrix containing the parameter estimates and associated statistics or null if Imsl.Stat.CategoricalGenLinModel.Solve (p. 380) has not been called.

Here, nCoef is the number of coefficients in the model. The statistics returned are as follows:

<table>
<thead>
<tr>
<th>Column</th>
<th>Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Coefficient estimate.</td>
</tr>
<tr>
<td>1</td>
<td>Estimated standard deviation of the estimated coefficient.</td>
</tr>
<tr>
<td>2</td>
<td>Asymptotic normal score for testing that the coefficient is zero.</td>
</tr>
<tr>
<td>3</td>
<td>ρ - value associated with the normal score in column 2.</td>
</tr>
</tbody>
</table>
Product

```csharp
virtual public double[] Product {get; }
```

The inverse of the Hessian times the gradient vector computed at the input parameter estimates. A `double` array of length `nCoeff` containing the inverse of the Hessian times the gradient vector computed at the input parameter estimates.

`nCoeff` is the number of coefficients in the model. This member function will call `Imsl.Stat.CategoricalGenLinModel.Solve (p. 380)` to get the product if the product has not already been computed.

UpperBound

```csharp
virtual public int UpperBound {set; }
```

Defines the upper bound on the sum of the number of distinct values taken on by each classification variable. An `int` scalar specifying the upper bound on the sum of the number of distinct values taken on by each classification variable.

Default: `UpperBound = 1`.

UpperEndpointColumn

```csharp
virtual public int UpperEndpointColumn {set; }
```

The column number in `x` that contains the upper endpoint of the observation interval for full interval and left interval observations. An `int` scalar which indicates the column number in `x` that contains the upper endpoint of the observation interval for full interval and left interval observations.

By default all observations are treated as "point" observations.

Constructor

```csharp
CategoricalGenLinModel
public CategoricalGenLinModel(double[,] x, Imsl.Stat.CategoricalGenLinModel.DistributionParameterModel model)
```

Constructs a new `CategoricalGenLinModel`.

Use one of the class members from the following table. The lower bound given in the table is the minimum possible value of the response variable:

<table>
<thead>
<tr>
<th>Model</th>
<th>Distribution</th>
<th>Function</th>
<th>Lower-bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Poisson</td>
<td>Exponential</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Negative Binomial</td>
<td>Logistic</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Logarithmic</td>
<td>Logistic</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Binomial</td>
<td>Logistic</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Binomial</td>
<td>Probit</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Binomial</td>
<td>Log-log</td>
<td>0</td>
</tr>
</tbody>
</table>

Let \( \gamma \) be the dot product of a row in the design matrix with the parameters (plus the fixed parameter, if used). Then, the functions used to model the distribution parameter are given by:
<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential</td>
<td>$e^\gamma$</td>
</tr>
<tr>
<td>Logistic</td>
<td>$e^\gamma/(1 + e^\gamma)$</td>
</tr>
<tr>
<td>Probit</td>
<td>$\Phi(\gamma)$ (where $\Phi$ is the normal cdf)</td>
</tr>
<tr>
<td>Log-log</td>
<td>$1 - e^{-\gamma}$</td>
</tr>
</tbody>
</table>

A **double** input matrix containing the data where the number of rows in the matrix is equal to the number of observations.

**model** An **int** scalar which specifies the distribution of the response variable and the function used to model the distribution parameter.

### Methods

**SetEffects**

```csharp
virtual public void SetEffects(int[] indef, int[] nvef)
```

Initializes an index vector to contain the column numbers in $x$ associated with each effect.

- **indef** contains the column numbers in $x$ that are associated with each effect. Member function `SetEffects(int [], int [] nvef [])` sets the number of variables associated with each effect in the model. The first `nvef [0]` elements of `indef` give the column numbers of the variables in the first effect. The next `nvef [0]` elements give the column numbers of the variables in the second effect, etc. By default this vector is not referenced.

- **nvef** contains the number of variables associated with each effect in the model. By default this vector is not referenced.

- **indef** An int vector of length $\sum_{k=0}^{nef-1} nvef [k]$ where `nef` is the number of effects in the model.

- **nvef** An int vector of length `nef` where `nef` is the number of effects in the model.

**System.ArgumentException** is thrown when an element of `indef` is less than 0 or greater than or equal to the number of columns of $x$ or if an element of `nvef` is less than or equal to 0

**SetInitialEstimates**

```csharp
virtual public void SetInitialEstimates(int init, double[] estimates)
```

Sets the initial parameter estimates option.

If this method is not called, `init` is set to 0.

<table>
<thead>
<tr>
<th>init</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unweighted linear regression is used to obtain initial estimates.</td>
</tr>
<tr>
<td>1</td>
<td>The nCoef, number of coefficients, elements of <code>estimates</code> contain initial estimates of the parameters. Use of this option requires that the user know nCoef beforehand.</td>
</tr>
</tbody>
</table>

`estimates` is used if `init = 1`. If this member function is not called, unweighted linear regression is used to obtain the initial estimates.
An input int indicating the desired initialization method for the initial estimates of the parameters.

An input double array of length nCoef containing the initial estimates of the parameters where nCoef is the number of estimated coefficients in the model.

System.ArgumentException is thrown when init is not in the range [0,1].

Solve

virtual public double[,] Solve()
Returns the parameter estimates and associated statistics for a CategoricalGenLinModel object.
Here, nCoef is the number of coefficients in the model. The statistics returned are as follows:

<table>
<thead>
<tr>
<th>Column</th>
<th>Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Coefficient estimate.</td>
</tr>
<tr>
<td>1</td>
<td>Estimated standard deviation of the estimated coefficient.</td>
</tr>
<tr>
<td>2</td>
<td>Asymptotic normal score for testing that the coefficient is zero.</td>
</tr>
<tr>
<td>3</td>
<td>ρ-value associated with the normal score in column 2.</td>
</tr>
</tbody>
</table>

Returns — An nCoef row by 4 column double matrix containing the parameter estimates and associated statistics.

Imsl.Stat.ClassificationVariableException is thrown when the number of values taken by each classification variable has been set by the user to be less than or equal to 1.

Imsl.Stat.ClassificationVariableLimitException is thrown when the sum of the number of distinct values taken on by each classification variable exceeds the maximum allowed, Imsl.Stat.CategoricalGenLinModel.UpperBound (p. 378).

Imsl.Stat.DeleteObservationsException is thrown if the number of observations to delete has grown too large.

Description

Reweighted least squares is used to compute (extended) maximum likelihood estimates in some generalized linear models involving categorized data. One of several models, including probit, logistic, Poisson, logarithmic, and negative binomial models, may be fit for input point or interval observations. (In the usual case, only point observations are observed.)

Let

\[ \gamma_i = w_i + x_i^T \beta = w_i + \eta_i \]

be the linear response where \( x_i \) is a design column vector obtained from a row of \( x \), \( \beta \) is the column vector of coefficients to be estimated, and \( w_i \) is a fixed parameter that may be input in \( x \). When some of the \( \gamma_i \) are infinite at the supremum of the likelihood, then extended maximum likelihood estimates are computed. Extended maximum likelihood is computed as the finite (but
nonunique) estimates $\hat{\beta}$ that optimize the likelihood containing only the observations with finite $\hat{\gamma}_i$. These estimates, when combined with the set of indices of the observations such that $\hat{\gamma}_i$ is infinite at the supremum of the likelihood, are called extended maximum estimates. When none of the optimal $\hat{\gamma}_i$ are infinite, extended maximum likelihood estimates are identical to maximum likelihood estimates. Extended maximum likelihood estimation is discussed in more detail by Clarkson and Jennrich (1991). In CategoricalGenLinModel, observations with potentially infinite $\hat{\eta}_i = x_i^T \hat{\beta}$ are detected and removed from the likelihood if Imsl.Stat.CategoricalGenLinModel.InfiniteEstimateMethod (p. 375) = 0. See below.

The models available in CategoricalGenLinModel are:

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Parameterization</th>
<th>Response PDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model0 (Poisson)</td>
<td>$\lambda = N \times e^{w+\eta}$</td>
<td>$f(y) = \lambda^y e^{-\lambda} / y!$</td>
</tr>
<tr>
<td>Model1 (Negative Binomial)</td>
<td>$\theta = \frac{e^{w+\eta}}{1+e^{w+\eta}}$</td>
<td>$f(y) = \frac{S+y-1}{y-1} \theta^S (1-\theta)^y$</td>
</tr>
<tr>
<td>Model2 (Logarithmic)</td>
<td>$\theta = \frac{e^{w+\eta}}{1+e^{w+\eta}}$</td>
<td>$f(y) = \frac{(1-\theta)^y}{y \ln \theta}$</td>
</tr>
<tr>
<td>Model3 (Logistic)</td>
<td>$\theta = \Phi(w+\eta)$</td>
<td>$f(y) = \binom{N}{y} \theta^y (1-\theta)^{N-y}$</td>
</tr>
<tr>
<td>Model4 (Probit)</td>
<td>$\theta = 1 - e^{-e^{w+\eta}}$</td>
<td>$f(y) = \binom{N}{y} \theta^y (1-\theta)^{N-y}$</td>
</tr>
</tbody>
</table>

Here $\Phi$ denotes the cumulative normal distribution, $N$ and $S$ are known parameters specified for each observation via column Imsl.Stat.CategoricalGenLinModel.OptionalDistributionParameterColumn (p. 377) of $x$, and $w$ is an optional fixed parameter specified for each observation via column Imsl.Stat.CategoricalGenLinModel.FixedParameterColumn (p. 374) of $x$. (By default $N$ is taken to be 1 for $model = 0, 3, 4$ and 5 and $S$ is taken to be 1 for $model = 1$. By default $w$ is taken to be 0.) Since the log-log model ($model = 5$) probabilities are not symmetric with respect to 0.5, quantitatively, as well as qualitatively, different models result when the definitions of "success" and "failure" are interchanged in this distribution. In this model and all other models involving $\theta$, $\theta$ is taken to be the probability of a "success."

Note that each row vector in the data matrix can represent a single observation; or, through the use of column Imsl.Stat.CategoricalGenLinModel.FrequencyColumn (p. 375) of the matrix $x$, each vector can represent several observations. Also note that classification variables and their products are easily incorporated into the models via the usual regression-type specifications.

**Computational Details**

For interval observations, the probability of the observation is computed by summing the probability distribution function over the range of values in the observation interval. For
right-interval observations, \( \Pr(Y \geq y) \) is computed as a sum based upon the equality
\( \Pr(Y \geq y) = 1 - \Pr(Y < y) \). Derivatives are similarly computed. The \texttt{CategoricalGenLinModel}
allows three types of interval observations. In full interval observations, both the lower and the
upper endpoints of the interval must be specified. For right-interval observations, only the lower
endpoint need be given while for left-interval observations, only the upper endpoint is given.

The computations proceed as follows:

- The input parameters are checked for consistency and validity.
- Estimates of the means of the “independent” or design variables are computed. The
  frequency of the observation in all but the binomial distribution model is taken from
  column \texttt{FrequencyColumn} of the data matrix \( x \). In binomial distribution models, the
  frequency is taken as the product of \( n = x[i, \text{OptionalDistributionParameterColumn}] \)
  and \( x[i, \text{FrequencyColumn}] \). In all cases these values default to 1. Means are computed
  as
  \[
  \bar{x} = \frac{\sum_{i} f_i x_i}{\sum_{i} f_i}
  \]
- If \( init = 0 \), initial estimates of the coefficients are obtained (based upon the observation
  intervals) as multiple regression estimates relating transformed observation probabilities
  to the observation design vector. For example, in the binomial distribution models, \( \theta \) for
  point observations may be estimated as
  \[
  \hat{\theta} = \frac{x[i, \text{LowerEndpointColumn}]}{x[i, \text{OptionalDistributionParameterColumn}]}
  \]
  and, when \( model = 3 \), the linear relationship is given by
  \[
  \left( \ln(\frac{\hat{\theta}}{1 - \hat{\theta}}) \right) \approx x\beta
  \]
  while if \( model = 4 \),
  \[
  \left( \Phi^{-1}(\hat{\theta}) = x\beta \right)
  \]
  For bounded interval observations, the midpoint of the interval is used for
  \( x[i, \text{Imsl.Stat.CategoricalGenLinModel.LowerEndpointColumn} \). Right-interval observations are not used in obtaining initial estimates when the
  distribution has unbounded support (since the midpoint of the interval is not defined).
  When computing initial estimates, standard modifications are made to prevent illegal
  operations such as division by zero.
  Regression estimates are obtained at this point, as well as later, by use of linear regression.
- Newton-Raphson iteration for the maximum likelihood estimates is implemented via
  iteratively reweighted least squares. Let
  \[
  \Psi(x^T \beta)
  \]
denote the log of the probability of the \(i\)-th observation for coefficients \(\beta\). In the least-squares model, the weight of the \(i\)-th observation is taken as the absolute value of the second derivative of

\[ \Psi(x_i^T \beta) \]

with respect to

\[ \gamma_i = x_i^T \beta \]

(times the frequency of the observation), and the dependent variable is taken as the first derivative \(\Psi\) with respect to \(\gamma_i\), divided by the square root of the weight times the frequency. The Newton step is given by

\[ \Delta \beta = \left( \sum_i |\Psi''(\gamma_i)|x_ix_i^T \right)^{-1} \sum_i \Psi'(\gamma_i)x_i \]

where all derivatives are evaluated at the current estimate of \(\gamma\), and \(\beta_{n+1} = \beta_n - \Delta \beta\). This step is computed as the estimated regression coefficients in the least-squares model. Step halving is used when necessary to ensure a decrease in the criterion.

- Convergence is assumed when the maximum relative change in any coefficient update from one iteration to the next is less than

  Imsl.Stat.CategoricalGenLinModel.ConvergenceTolerance (p. 373) or when the relative change in the log-likelihood from one iteration to the next is less than

  ConvergenceTolerance/100. Convergence is also assumed after

  Imsl.Stat.CategoricalGenLinModel.MaxIterations (p. 376) or when step halving leads to a step size of less than .0001 with no increase in the log-likelihood.

- For interval observations, the contribution to the log-likelihood is the log of the sum of the probabilities of each possible outcome in the interval. Because the distributions are discrete, the sum may involve many terms. The user should be aware that data with wide intervals can lead to expensive (in terms of computer time) computations.

- If InfiniteEstimateMethod is set to 0, then the methods of Clarkson and Jennrich (1991) are used to check for the existence of infinite estimates in

  \[ \eta_i = x_i^T \beta \]

As an example of a situation in which infinite estimates can occur, suppose that observation \(j\) is right censored with \(t_j > 15\) in a logistic model. If design matrix \(x\) is such that \(x_{jm} = 1\) and \(x_{im} = 0\) for all \(i \neq j\), then the optimal estimate of \(\beta_m\) occurs at

\[ \hat{\beta}_m = \infty \]
leading to an infinite estimate of both $\beta_m$ and $\eta_j$. In CategoricalGenLinModel, such estimates may be "computed."

In all models fit by CategoricalGenLinModel, infinite estimates can only occur when the optimal estimated probability associated with the left- or right-censored observation is 1. If InfiniteEstimateMethod is set to 0, left- or right-censored observations that have estimated probability greater than 0.995 at some point during the iterations are excluded from the log-likelihood, and the iterations proceed with a log-likelihood based upon the remaining observations. This allows convergence of the algorithm when the maximum relative change in the estimated coefficients is small and also allows for the determination of observations with infinite estimates.

$$\eta_i = x_i^T \beta$$

At convergence, linear programming is used to ensure that the eliminated observations have infinite $\eta_i$. If some (or all) of the removed observations should not have been removed (because their estimated $\eta_i$'s must be finite), then the iterations are restarted with a log-likelihood based upon the finite $\eta_i$ observations. See Clarkson and Jennrich (1991) for more details.

When InfiniteEstimateMethod is set to 1, no observations are eliminated during the iterations. In this case, when infinite estimates occur, some (or all) of the coefficient estimates $\hat{\beta}$ will become large, and it is likely that the Hessian will become (numerically) singular prior to convergence.

When infinite estimates for the $\eta_i$ are detected, linear regression (see Chapter 2, Regression:) is used at the convergence of the algorithm to obtain unique estimates $\hat{\beta}$. This is accomplished by regressing the optimal $\eta_i$ or the observations with finite $\eta$ against $x\beta$, yielding a unique $\hat{\beta}$ (by setting coefficients $\hat{\beta}$ that are linearly related to previous coefficients in the model to zero). All of the final statistics relating to $\hat{\beta}$ are based upon these estimates.

Residuals are computed according to methods discussed by Pregibon (1981). Let $\ell_i(\gamma_i)$ denote the log-likelihood of the $i$-th observation evaluated at $\gamma_i$. Then, the standardized residual is computed as

$$r_i = \frac{\ell_i'(\hat{\gamma}_i)}{\sqrt{\ell_i''(\hat{\gamma}_i)}}$$

where $\hat{\gamma}_i$ is the value of $\gamma_i$ when evaluated at the optimal $\hat{\beta}$ and the derivatives here (and only here) are with respect to $\gamma$ rather than with respect to $\beta$. The denominator of this expression is used as the "standard error of the residual" while the numerator is the "raw" residual.

Following Cook and Weisberg (1982), we take the influence of the $i$-th observation to be

$$\ell_i'(\hat{\gamma}_i)\ell_i''(\hat{\gamma}_i)^{-1}\ell_i'(\hat{\gamma}_i)$$
This quantity is a one-step approximation to the change in the estimates when the \( i \)-th observation is deleted. Here, the partial derivatives are with respect to \( \beta \).

**Programming Notes**

- Classification variables are specified via `Imsl.Stat.CategoricalGenLinModel.ClassificationVariableColumn` (p. 373). Indicator or dummy variables are created for the classification variables.
- To enhance precision "centering" of covariates is performed if `Imsl.Stat.CategoricalGenLinModel.ModelIntercept` (p. 376) is set to 1 and (number of observations) - (number of rows in \( z \) missing one or more values) > 1. In doing so, the sample means of the design variables are subtracted from each observation prior to its inclusion in the model. On convergence the intercept, its variance and its covariance with the remaining estimates are transformed to the uncentered estimate values.
- Two methods for specifying a binomial distribution model are possible. In the first method, \( x[i,\text{FrequencyColumn}] \) contains the frequency of the observation while \( x[i,\text{LowerEndpointColumn}] \) is 0 or 1 depending upon whether the observation is a success or failure. In this case, \( N = x[i,\text{OptionalDistributionParameterColumn}] \) is always 1. The model is treated as repeated Bernoulli trials, and interval observations are not possible.

A second method for specifying binomial models is to use \( x[i,\text{LowerEndpointColumn}] \) to represent the number of successes in the \( x[i,\text{OptionalDistributionParameterColumn}] \) trials. In this case, \( x[i,\text{FrequencyColumn}] \) will usually be 1, but it may be greater than 1, in which case interval observations are possible.

Note that the `Imsl.Stat.CategoricalGenLinModel.Solve` (p. 380) method must be called before using any property as a right operand, otherwise the value is null.

**Example 1: Example: Mortality of beetles.**

The first example is from Prentice (1976) and involves the mortality of beetles after exposure to various concentrations of carbon disulphide. Both a logit and a probit fit are produced for linear model \( \mu + \beta x \). The data is given as

<table>
<thead>
<tr>
<th>Covariate(x)</th>
<th>N</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.755</td>
<td>62</td>
<td>18</td>
</tr>
<tr>
<td>1.784</td>
<td>56</td>
<td>28</td>
</tr>
<tr>
<td>1.811</td>
<td>63</td>
<td>52</td>
</tr>
<tr>
<td>1.836</td>
<td>59</td>
<td>53</td>
</tr>
<tr>
<td>1.861</td>
<td>62</td>
<td>61</td>
</tr>
<tr>
<td>1.883</td>
<td>60</td>
<td>60</td>
</tr>
</tbody>
</table>
using System;
using Imsl.Math;
using Imsl.Stat;

class CategoricalGenLinModelEx1
{
    public static void Main(string[] args)
    {
        // Set up a PrintMatrix object for later use.
        PrintMatrixFormat mf;
        PrintMatrix p;
        p = new PrintMatrix();
        mf = new PrintMatrixFormat();
        mf.SetNoRowLabels();
        mf.SetNoColumnLabels();
        mf.NumberFormat = "0.0000";
        double[,] x = {{1.69, 59.0, 6.0},
                        {1.724, 60.0, 13.0},
                        {1.755, 62.0, 18.0},
                        {1.784, 56.0, 28.0},
                        {1.811, 63.0, 52.0},
                        {1.836, 59.0, 53.0},
                        {1.861, 62.0, 61.0},
                        {1.883, 60.0, 60.0}};
        CategoricalGenLinModel CATGLM3, CATGLM4;
        // MODEL3
        CATGLM3 = new CategoricalGenLinModel(x,
                                              CategoricalGenLinModel.DistributionParameterModel.Model3);
        CATGLM3.LowerEndpointColumn = 2;
        CATGLM3.OptionalDistributionParameterColumn = 1;
        CATGLM3.InfiniteEstimateMethod = 1;
        CATGLM3.ModelIntercept = 1;
        int[] nvef = new int[]{1};
        int[] indef = new int[]{0};
        CATGLM3.SetEffects(indef, nvef);
        CATGLM3.UpperBound = 1;
        Console.Out.WriteLine("MODEL3");
        p.SetTitle("Coefficient Statistics");
        p.Print(mf, CATGLM3.Solve());
        Console.Out.WriteLine("Log likelihood " + CATGLM3.OptimizedCriterion);
        p.SetTitle("Asymptotic Coefficient Covariance");
        p.SetMatrixType(PrintMatrix.MatrixType.UpperTriangular);
        p.Print(mf, CATGLM3.CovarianceMatrix);
        p.SetMatrixType(PrintMatrix.MatrixType.Full);
        p.SetTitle("Case Analysis");
        p.Print(mf, CATGLM3.CaseAnalysis);
        p.SetTitle("Last Coefficient Update");
        p.Print(CATGLM3.LastParameterUpdates);
        p.SetTitle("Covariate Means");
    }
}

CategoricalGenLinModel Class IMSL C# Numerical Library
p.Print(CATGLM3.DesignVariableMeans);
p.SetTitle("Observation Codes");
p.Print(CATGLM3.ExtendedLikelihoodObservations);
Console.Out.WriteLine("Number of Missing Values " + CATGLM3.NRowsMissing);

// MODEL4
CATGLM4 = new CategoricalGenLinModel(x,
    CategoricalGenLinModel.DistributionParameterModel.Model4);
CATGLM4.LowerEndpointColumn = 2;
CATGLM4.OptionalDistributionParameterColumn = 1;
CATGLM4.InfiniteEstimateMethod = 1;
CATGLM4.ModelIntercept = 1;
CATGLM4.SetEffects(indef, nvef);
CATGLM4.UpperBound = 1;
CATGLM4.Solve();

Console.Out.WriteLine("\nMODEL4");
Console.Out.WriteLine("Log likelihood " + CATGLM4.OptimizedCriterion);
p.SetTitle("Coefficient Statistics");
p.Print(mf, CATGLM4.Parameters); } }

Output

MODEL3

Coefficient Statistics

-60.7568 5.1876 -11.7118 0.0000
34.2985 2.9164 11.7607 0.0000

Log likelihood -18.778179042334
Asymptotic Coefficient Covariance

26.9117 -15.1243
8.5052

Case Analysis

0.0677 2.5934 1.7916 0.2674 1.4475
0.1644 3.1390 2.8706 0.3470 1.0935
0.3629 -4.4976 3.7860 0.3108 -1.1879
0.6063 -5.9517 3.6562 0.2322 -1.6279
0.7954 1.8901 3.2020 0.2688 0.5903
0.9016 -0.1949 2.2878 0.2380 -0.0852
0.9558 1.7434 1.6193 0.1976 0.1076
0.9787 1.2783 1.1185 0.1382 0.1142
0.9787 1.2783 1.1185 0.1382 0.1142

Last Coefficient Update

0
0 1.85793237772546E-07
1 1.33163785436183E-05

Categorical and Discrete Data Analysis  CategoricalGenLinModel Class ● 387
**Example 2: Example: Poisson Model.**

In this example, the following data illustrate the Poisson model when all types of interval data are present. The example also illustrates the use of classification variables and the detection of potentially infinite estimates (which turn out here to be finite). These potential estimates lead to the two iteration summaries. The input data is

<table>
<thead>
<tr>
<th>ilt</th>
<th>irt</th>
<th>icen</th>
<th>Class 1</th>
<th>Class 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

A linear model $\mu + \beta_1 x_1 + \beta_2 x_2$ is fit where $x_1 = 1$ if the Class 1 variable is 0, $x_1 = 1$, otherwise, and the $x_2$ variable is similarly defined.

```csharp
using System;
using Imsl.Math;
using Imsl.Stat;

class CategoricalGenLinModelEx2
{
    public static void Main(string[] args)
    {
```
```csharp
// Set up a PrintMatrix object for later use.
PrintMatrixFormat mf;
PrintMatrix p;
p = new PrintMatrix();
mf = new PrintMatrixFormat();
mf.SetNoRowLabels();
mf.SetNoColumnLabels();
    mf.NumberFormat = "0.0000";

double[,] x = {
    {0.0, 5.0, 0.0, 1.0, 0.0},
    {9.0, 4.0, 3.0, 0.0, 0.0},
    {0.0, 4.0, 1.0, 0.0, 0.0},
    {9.0, 0.0, 2.0, 1.0, 1.0},
    {0.0, 1.0, 0.0, 0.0, 1.0}};
CategoricalGenLinModel CATGLM;
CATGLM = new CategoricalGenLinModel(x,
    CategoricalGenLinModel.DistributionParameterModel.Model0);
CATGLM.UpperEndpointColumn = 0;
CATGLM.LowerEndpointColumn = 1;
CATGLM.OptionalDistributionParameterColumn = 1;
CATGLM.CensorColumn = 2;
CATGLM.InfiniteEstimateMethod = 0;
CATGLM.ModelIntercept = 1;
int[] indcl = new int[] {3, 4};
CATGLM.ClassificationVariableColumn = indcl;
int[] nvef = new int[] {1, 1};
int[] indef = new int[] {3, 4};
CATGLM.SetEffects(indef, nvef);
CATGLM.UpperBound = 4;

p.SetTitle("Coefficient Statistics");
p.Print(mf, CATGLM.Solve());
Console.Out.WriteLine("Log likelihood " + CATGLM.OptimizedCriterion);
p.SetTitle("Asymptotic Coefficient Covariance");
p.SetMatrixType(PrintMatrix.MatrixType.UpperTriangular);
p.Print(mf, CATGLM.CovarianceMatrix);
p.SetMatrixType(PrintMatrix.MatrixType.Full);
p.SetTitle("Case Analysis");
p.Print(mf, CATGLM.CaseAnalysis);
p.SetTitle("Last Coefficient Update");
p.Print(CATGLM.LastParameterUpdates);
p.SetTitle("Covariate Means");
p.Print(CATGLM.DesignVariableMeans);
p.SetTitle("Distinct Values For Each Class Variable");
p.Print(CATGLM.ClassificationVariableValues);
Console.Out.WriteLine("Number of Missing Values " + CATGLM.NRowsMissing);
};
```
Output

Coefficient Statistics
-0.5488  1.1713  -0.4685  0.6395
0.5488   0.6098   0.8999  0.3684
0.5488   1.0825   0.5069  0.6123

Log likelihood -3.11463849257844
Asymptotic Coefficient Covariance
1.3719  -0.3719  -1.1719
0.3719   0.1719   0.3515
1.1719

Case Analysis
5.0000   0.0000  2.2361  1.0000  0.0000
6.9246  -0.4122  2.1078  0.7636  -0.1955
6.9246   0.4122  1.1727  0.2364  0.3515
0.0000   0.0000  0.0000  0.0000  NaN
1.0000   0.0000  1.0000  1.0000  0.0000

Last Coefficient Update
0
0  -2.84092901922464E-07
1   3.53822215072981E-10
2   7.09878432577707E-07

Covariate Means
0
0  0.6
1  0.6
2  0

Distinct Values For Each Class Variable
0
0  0
1  1
2  0
3  1

Number of Missing Values 0

CategoricalGenLinModel.DistributionParameterModel
Enumeration

Indicates the function used to model the distribution parameter.

**Fields**

Model0
public Imsl.Stat.CategoricalGenLinModel.DistributionParameterModel Model0
Indicates an exponential function is used to model the distribution parameter. The distribution of the response variable is Poisson. The lower bound of the response variable is 0.

Model1
public Imsl.Stat.CategoricalGenLinModel.DistributionParameterModel Model1
Indicates a logistic function is used to model the distribution parameter. The distribution of the response variable is negative Binomial. The lower bound of the response variable is 0.

Model2
Indicates a logistic function is used to model the distribution parameter. The distribution of the response variable is Logarithmic. The lower bound of the response variable is 1.

Model3
Indicates a logistic function is used to model the distribution parameter. The distribution of the response variable is Binomial. The lower bound of the response variable is 0.

Model4
public Imsl.Stat.CategoricalGenLinModel.DistributionParameterModel Model4
Indicates a probit function is used to model the distribution parameter. The distribution of the response variable is Binomial. The lower bound of the response variable is 0.

Model5
Indicates a log-log function is used to model the distribution parameter. The distribution of the response variable is Binomial. The lower bound of the response variable is 0.
Chapter 16: Nonparametric Statistics

Types

\textit{class} SignTest ................................................................. 394
\textit{class} WilcoxonRankSum .................................................. 397

Usage Notes

Much of what is considered nonparametric statistics is included in other chapters. Topics of possible interest in other chapters are: nonparametric measures of location and scale (see "Basic Statistics"), nonparametric measures in a contingency table (see "Categorical and Discrete Data Analysis"), measures of correlation in a contingency table (see "Correlation and Covariance"), and tests of goodness of fit and randomness (see "Tests of Goodness of Fit and Randomness").

Missing Values

Most classes described in this chapter automatically handle missing values (NaN, "Not a Number"; see \texttt{Double.NaN}).

Tied Observations

The \texttt{WilcoxonRankSum} class described in this chapter contains a set method, \texttt{setFuzz}. Observations that are within fuzz of each other in absolute value are said to be tied. If \texttt{fuzz} = 0.0, observations must be identically equal before they are considered to be tied. Other positive values of fuzz allow for numerical imprecision or roundoff error.
SignTest Class

Performs a sign test.

```csharp
public class Imsl.Stat.SignTest
```

**Properties**

**NumPositiveDev**
```csharp
public int NumPositiveDev {get; }
```
Returns the number of positive differences. A scalar int containing the number of positive differences \(x[j-1] \text{-Percentile}\) for \(j = 1, 2, \ldots, x.Length\).

**NumZeroDev**
```csharp
public int NumZeroDev {get; }
```
Returns the number of zero differences. A scalar int containing the number of zero differences (ties) \(x[j-1] \text{-Percentile}\) for \(j = 1, 2, \ldots, x.Length\).

**Percentage**
```csharp
public double Percentage {get; set; }
```
The percentage percentile of the population. A double scalar containing the value in the range \((0, 1)\).

Default: Percentage = 0.5.

**Percentile**
```csharp
public double Percentile {get; set; }
```
The hypothesized percentile of the population. A double scalar containing the hypothesized percentile of the population from which \(x\) was drawn.

Default: Percentile = 0.0

**Constructor**

```csharp
public SignTest(double[] x)
```
Constructor for SignTest.

- \(x\) A double array containing the data.

**Method**

**Compute**
public double Compute()
Perform a sign test.

Call this value probability. If using default values, the null hypothesis is that the median equals 0.0.

Returns — A double scalar containing the Binomial probability of NumPositiveDev or more positive differences in x.length - number of zero differences trials.

Description

Class SignTest tests hypotheses about the proportion \( p \) of a population that lies below a value \( q \), where \( p \) and \( q \) corresponds to the Percentage and Percentile properties, respectively. In continuous distributions, this can be a test that \( q \) is the 100 \( p \)-th percentile of the population from which \( x \) was obtained. To carry out testing, SignTest tallies the number of values above \( q \) in the number of positive differences \( x[j - 1] - \text{Percentile} \) for \( j = 1, 2, \ldots, \text{x.length} \). The binomial probability of the number of values above \( q \) in the number of positive differences \( x[j - 1] - \text{Percentile} \) for \( j = 1, 2, \ldots, \text{x.length} \) or more values above \( q \) is then computed using the proportion \( p \) and the sample size in \( x \) (adjusted for the missing observations and ties).

Hypothesis testing is performed as follows for the usual null and alternative hypotheses:

- \( H_0 : \text{Pr}(x \leq q) \geq p \) (the \( p \)-th quantile is at least \( q \))
  \( H_1 : \text{Pr}(x \leq q) < p \)
  Reject \( H_0 \) if probability is less than or equal to the significance level.

- \( H_0 : \text{Pr}(x \leq q) \leq p \) (the \( p \)-th quantile is at least \( q \))
  \( H_1 : \text{Pr}(x \leq q) > p \)
  Reject \( H_0 \) if probability is greater than or equal to 1 minus the significance level.

- \( H_0 : \text{Pr}(x = q) = p \) (the \( p \)-th quantile is \( q \))
  \( H_1 : \text{Pr}(x \leq q) < p \) or \( \text{Pr}(x \leq q) > p \)
  Reject \( H_0 \) if probability is less than or equal to half the significance level or greater than or equal to 1 minus half the significance level.

The assumptions are as follows:

- They are independent and identically distributed.
- Measurement scale is at least ordinal; i.e., an ordering less than, greater than, and equal to exists in the observations.

Many uses for the sign test are possible with various values of \( p \) and \( q \). For example, to perform a matched sample test that the difference of the medians of \( y \) and \( z \) is 0.0, let \( p = 0.5, q = 0.0 \), and \( x_i = y_i - z_i \) in matched observations \( y \) and \( z \). To test that the median difference is \( c \), let \( q = c \).
**Example 1: Sign Test**

This example tests the hypothesis that at least 50 percent of a population is negative. Because $0.18 < 0.95$, the null hypothesis at the 5-percent level of significance is not rejected.

```csharp
using System;
using Imsl.Stat;

public class SignTestEx1
{
    public static void Main(String[] args)
    {
        double[] x = new double[] { 92.0, 139.0, -6.0, 10.0, 81.0, -11.0, 45.0, -25.0, -4.0, 22.0, 2.0, 41.0, 13.0, 8.0, 33.0, 45.0, -33.0, -45.0, -12.0};
        SignTest st = new SignTest(x);
        Console.Out.WriteLine("Probability = " + st.Compute().ToString("0.000000");
    }
}
```

**Output**

Probability = 0.179642

**Example 2: Sign Test**

This example tests the null hypothesis that at least 75 percent of a population is negative. Because $0.923 < 0.95$, the null hypothesis at the 5-percent level of significance is rejected.

```csharp
using System;
using Imsl.Stat;

public class SignTestEx2
{
    public static void Main(String[] args)
    {
        double[] x = new double[] { 92.0, 139.0, -6.0, 10.0, 81.0, -11.0, 45.0, -25.0, -4.0, 22.0, 2.0, 41.0, 13.0, 8.0, 33.0, 45.0, -33.0, -45.0, -12.0};
        SignTest st = new SignTest(x);
    }
}
```
st.Percentage = 0.75;
st.Percentile = 0.0;
Console.Out.WriteLine("Probability = " + st.Compute().ToString("0.000000"));
Console.Out.WriteLine("Number of positive deviations = " + st.NumPositiveDev);
Console.Out.WriteLine("Number of ties = " + st.NumZeroDev);
}

Output

Probability = 0.922543
Number of positive deviations = 12
Number of ties = 0

WilcoxonRankSum Class

Performs a Wilcoxon rank sum test.

public class Imsl.Stat.WilcoxonRankSum

Constructor

WilcoxonRankSum
    public WilcoxonRankSum(double[] x, double[] y)
    Constructor for WilcoxonRankSum.
    x  A double array containing the first sample.
    y  A double array containing the second sample.

Methods

Compute
    public double Compute()
    Performs a Wilcoxon rank sum test.
    Returns — A double scalar containing the two-sided p-value for the Wilcoxon rank sum statistic that is computed with average ranks used in the case of ties.

GetStatistics
    public double[] GetStatistics()
Returns the statistics.
The statistics are as follows:

<table>
<thead>
<tr>
<th>Row</th>
<th>Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Wilcoxon $W$ statistic (the sum of the ranks of the $x$ observations) adjusted for ties in such a manner that $W$ is as small as possible</td>
</tr>
<tr>
<td>1</td>
<td>$2 \times E(W) - W$, where $E(W)$ is the expected value of $W$</td>
</tr>
<tr>
<td>2</td>
<td>probability of obtaining a statistic less than or equal to $\min{W, 2 \times E(W) - W}$</td>
</tr>
<tr>
<td>3</td>
<td>$W$ statistic adjusted for ties in such a manner that $W$ is as large as possible</td>
</tr>
<tr>
<td>4</td>
<td>$2 \times E(W) - W$, where $E(W)$ is the expected value of $W$, adjusted for ties in such a manner that $W$ is as large as possible</td>
</tr>
<tr>
<td>5</td>
<td>probability of obtaining a statistic less than or equal to $\min{W, 2 \times E(W) - W}$, adjusted for ties in such a manner that $W$ is as large as possible</td>
</tr>
<tr>
<td>6</td>
<td>$W$ statistic with average ranks used in case of ties</td>
</tr>
<tr>
<td>7</td>
<td>estimated standard error of Row 6 under the null hypothesis of no difference</td>
</tr>
<tr>
<td>8</td>
<td>standard normal score associated with Row 6</td>
</tr>
<tr>
<td>9</td>
<td>two-sided p-value associated with Row 8</td>
</tr>
</tbody>
</table>

Returns — A **double** array of length 10 containing statistics.

SetFuzz

```csharp
public void SetFuzz(double fuzz)
```

Sets the nonnegative constant used to determine ties in computing ranks in the combined samples.

A tie is declared when two observations in the combined sample are within `fuzz` of each other. Default: $fuzz = 100 \times 2.220446049250313e - 16 \times \max(|x_{i1}|, |x_{j2}|)$

**fuzz** A **double** scalar containing the nonnegative constant used to determine ties in computing ranks in the combined samples.

**Description**

Class **WilcoxonRankSum** performs the Wilcoxon rank sum test for identical population distribution functions. The Wilcoxon test is a linear transformation of the Mann-Whitney $U$ test. If the difference between the two populations can be attributed solely to a difference in location, then the Wilcoxon test becomes a test of equality of the population means (or medians) and is the nonparametric equivalent of the two-sample $t$-test. Class **WilcoxonRankSum** obtains ranks in the combined sample after first eliminating missing values from the data. The rank sum statistic is then computed as the sum of the ranks in the $x$ sample. Three methods for handling ties are used. (A tie is counted when two observations are within `fuzz` of each
other.) Method 1 uses the largest possible rank for tied observations in the smallest sample, while Method 2 uses the smallest possible rank for these observations. Thus, the range of possible rank sums is obtained.

Method 3 for handling tied observations between samples uses the average rank of the tied observations. Asymptotic standard normal scores are computed for the \( W \) score (based on a variance that has been adjusted for ties) when average ranks are used (see Conover 1980, p. 217), and the probability associated with the two-sided alternative is computed.

**Hypothesis Tests**

In each of the following tests, the first line gives the hypothesis (and its alternative) under the assumptions 1 to 3 below, while the second line gives the hypothesis when assumption 4 is also true. The rejection region is the same for both hypotheses and is given in terms of Method 3 for handling ties. If another method for handling ties is desired, another output statistic, \( \text{stat}[0] \) or \( \text{stat}[3] \), should be used, where \( \text{stat} \) is the array containing the statistics returned from the \( \text{getStatistics} \) method.

<table>
<thead>
<tr>
<th>Test</th>
<th>Null Hypothesis</th>
<th>Alternative Hypothesis</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( H_0: \Pr(x_1 &lt; x_2) = 0.5 ) ( H_0: E(x_1) = E(x_2) )</td>
<td>( H_1: \Pr(x_1 &lt; x_2) \neq 0.5 ) ( H_1: E(x_1) \neq E(x_2) )</td>
<td>Reject if ( \text{stat}[9] ) is less than the significance level of the test. Alternatively, reject the null hypothesis if ( \text{stat}[6] ) is too large or too small.</td>
</tr>
<tr>
<td>2</td>
<td>( H_0: \Pr(x_1 &lt; x_2) \leq 0.5 ) ( H_0: E(x_1) \geq E(x_2) )</td>
<td>( H_1: \Pr(x_1 &lt; x_2) \neq 0.5 ) ( H_1: E(x_1) &lt; E(x_2) )</td>
<td>Reject if ( \text{stat}[6] ) is too small</td>
</tr>
<tr>
<td>3</td>
<td>( H_0: \Pr(x_1 &lt; x_2) \geq 0.5 ) ( H_0: E(x_1) \leq E(x_2) )</td>
<td>( H_1: \Pr(x_1 &lt; x_2) &lt; 0.5 ) ( H_1: E(x_1) &gt; E(x_2) )</td>
<td>Reject if ( \text{stat}[6] ) is too large</td>
</tr>
</tbody>
</table>

**Assumptions**

- \( x \) and \( y \) contain random samples from their respective populations.
- All observations are mutually independent.
- The measurement scale is at least ordinal (i.e., an ordering less than, greater than, or equal to exists among the observations).
- If \( f(x) \) and \( g(y) \) are the distribution functions of \( x \) and \( y \), then \( g(y) = f(x + c) \) for some constant \( c \) (i.e., the distribution of \( y \) is, at worst, a translation of the distribution of \( x \)).

Tables of critical values of the \( W \) statistic are given in the references for small samples.

**Example 1: Wilcoxon Rank Sum Test**

The following example is taken from Conover (1980, p. 224). It involves the mixing time of two
mixing machines using a total of 10 batches of a certain kind of batter, five batches for each machine. The null hypothesis is not rejected at the 5-percent level of significance.

using System;
using Imsl;
using Imsl.Stat;

public class WilcoxonRankSumEx1
{
    public static void Main(String[] args)
    {
        double[] x = new double[]{7.3, 6.9, 7.2, 7.8, 7.2};
        double[] y = new double[]{7.4, 6.8, 6.9, 6.7, 7.1};
        WilcoxonRankSum wilcoxon = new WilcoxonRankSum(x, y);
        Console.Out.WriteLine("p-value = " + wilcoxon.Compute().ToString("0.0000");
    }
}

Output

p-value = 0.1412
Imsl.Stat.WilcoxonRankSum: "x.length" = 5 and "y.length" = 5.
Both sample sizes, "x.length" and "y.length", are less than 25.
Significance levels should be obtained from tabled values.
Imsl.Stat.WilcoxonRankSum: At least one tie is detected between the samples.

Example 2: Wilcoxon Rank Sum Test

The following example uses the same data as in example 1. Now, all the statistics are displayed.

using System;
using Imsl.Stat;
using PrintMatrix = Imsl.Math.PrintMatrix;

public class WilcoxonRankSumEx2
{
    public static void Main(String[] args)
    {
        double[] x = new double[]{7.3, 6.9, 7.2, 7.8, 7.2};
        double[] y = new double[]{7.4, 6.8, 6.9, 6.7, 7.1};
        String[] labels = new String[]{
            "Wilcoxon W statistic ..................",
            "2*E(W) - W ..............................",
            "p-value .............................",
            "Adjusted Wilcoxon statistic ..........",
            "Adjusted 2*E(W) - W ...................",
            "Adjusted p-value ......................",
            "W statistics for averaged ranks......."
        };
        //...
    }
}
WilcoxonRankSum wilcoxon = new WilcoxonRankSum(x, y);
wilcoxon.Compute();
double[] stat = wilcoxon.GetStatistics();

for (int i = 0; i < 10; i++)
{
    Console.Out.WriteLine((labels[i] + " " + stat[i].ToString("0.000")));
}

---

Output

Wilcoxon W statistic ......................... 34.000
2*E(W) − W ................................... 21.000
p-value .................................... 0.110
Adjusted Wilcoxon statistic .................. 35.000
Adjusted 2*E(W) − W ......................... 20.000
Adjusted p-value ............................ 0.075
W statistics for averaged ranks ............. 34.500
Standard error of W (averaged ranks) ....... 4.758
Standard normal score of W (averaged ranks) 1.471
Two-sided p-value of W (averaged ranks) ... 0.141

Imsl.Stat.WilcoxonRankSum: "x.length" = 5 and "y.length" = 5.
Both sample sizes, "x.length" and "y.length", are less than 25.
Significance levels should be obtained from tabled values.
Imsl.Stat.WilcoxonRankSum: At least one tie is detected between the samples.
Chapter 17: Tests of Goodness of Fit

Types

\texttt{class ChiSquaredTest} ................................................................. 403
\texttt{class NormalityTest} ................................................................. 408

Usage Notes

The classes in this chapter are used to test for goodness of fit. The goodness-of-fit tests are described in Conover (1980). There is a goodness-of-fit test for general distributions and a chi-squared test. The user supplies the hypothesized cumulative distribution function for the test. There is a class that can be used to test specifically for the normal distribution.

The chi-squared goodness-of-fit test may be used with discrete as well as continuous distributions. The chi-squared goodness-of-fit test allows for missing values (NaN, not a number) in the input data.

ChiSquaredTest Class

Chi-squared goodness-of-fit test.

\texttt{public class Imsl.Stat.ChiSquaredTest}

Properties

ChiSquared

\hspace{1cm} \texttt{public double ChiSquared \{get; \}}
The chi-squared statistic. A double containing the chi-squared statistic.

DegreesOfFreedom

public double DegreesOfFreedom {get; }

Returns the degrees of freedom in chi-squared. A double containing the degrees of freedom in the chi-squared statistic.

P

public double P {get; }

The p-value for the chi-squared statistic. A double containing the p-value for the chi-squared statistic.

Constructors

ChiSquaredTest

public ChiSquaredTest(Imsl.Stat.ICdfFunction cdf, double[] cutpoints, int nParameters)

Constructor for the Chi-squared goodness-of-fit test.

cdf Object that implements the ICdfFunction interface.
cutpoints A double array containing the cutpoints.
nParameters A int which specifies the number of parameters estimated in computing the Cdf.

Imsl.Stat.NotCDFException is thrown if the function cdf.CdfFunction is not a valid CDF.

ChiSquaredTest

public ChiSquaredTest(Imsl.Stat.ICdfFunction cdf, int nCutpoints, int nParameters)

Constructor for the Chi-squared goodness-of-fit test

cdf Object that implements the ICdfFunction interface.
nCutpoints A int which specifies the number of cutpoints.
nParameters A int which specifies the number of parameters estimated in computing the Cdf.

Imsl.Stat.NotCDFException is thrown if the function cdf.CdfFunction is not a valid CDF.

Imsl.Stat.DidNotConvergeException is thrown if the iteration to find the inverse of the CDF did not converge. The inverse CDF is needed to compute the cutpoints.
Methods

GetCellCounts
    public double[] GetCellCounts()
    Returns the cell counts.
    Returns — A double array which contains the number of actual observations in each cell.

GetCutpoints
    public double[] GetCutpoints()
    Returns the cutpoints.
    The intervals defined by the cutpoints are such that the lower endpoint is not included while the upper endpoint is included in the interval.
    Returns — A double array which contains the cutpoints.

GetExpectedCounts
    public double[] GetExpectedCounts()
    Returns the expected counts.
    Returns — A double array which contains the number of expected observations in each cell.

SetCutpoints
    public void SetCutpoints(double[] cutpoints)
    Sets the cutpoints.
    The intervals defined by the cutpoints are such that the lower endpoint is not included while the upper endpoint is included in the interval.
    cutpoints — A double array which contains the cutpoints.

SetRange
    public void SetRange(double lower, double upper)
    Sets endpoints of the range of the distribution.
    Points outside of the range are ignored so that distributions conditional on the range can be used. In this case, the point lower is excluded from the first interval, but the point upper is included in the last interval.
    By default, a range on the whole real line is used.
    lower — A double which specifies the lower range limit.
    upper — A double which specifies the upper range limit.

Update
    public void Update(double[] x, double[] freq)
    Adds new observations to the test.
    x — A double array which contains the new observations to be added to the test.
freq  A double array which contains the frequencies of the corresponding new observations in x.

Update
   public void Update(double x, double freq)
   Adds a new observation to the test.
   
   x  A double which specifies the new observation to be added to the test.
   freq  A double which specifies the frequency of the new observation, x.

Description

ChiSquaredTest performs a chi-squared goodness-of-fit test that a random sample of observations is distributed according to a specified theoretical cumulative distribution. The theoretical distribution, which may be continuous, discrete, or a mixture of discrete and continuous distributions, is specified via a user-defined function \( F \) where \( F \) implements ICdfFunction. Because the user is allowed to specify a range for the observations in the SetRange method, a test that is conditional upon the specified range is performed.

ChiSquaredTest can be constructed in two different ways. The intervals can be specified via the array cutpoints. Otherwise, the number of cutpoints can be given and equiprobable intervals computed by the constructor. The observations are divided into these intervals. Regardless of the method used to obtain them, the intervals are such that the lower endpoint is not included in the interval while the upper endpoint is always included. The user should determine the cutpoints when the cumulative distribution function has discrete elements since ChiSquaredTest cannot determine them in this case.

By default, the lower and upper endpoints of the first and last intervals are \(-\infty\) and \(+\infty\), respectively. The method SetRange can be used to change the range.

A tally of counts is maintained for the observations in \( x \) as follows:

If the cutpoints are specified by the user, the tally is made in the interval to which \( x_i \) belongs, using the user-specified endpoints.

If the cutpoints are determined by the class then the cumulative probability at \( x_i, F(x_i) \), is computed using Cdf.

The tally for \( x_i \) is made in interval number \( \lfloor mF(x) + 1 \rfloor \), where \( m \) is the number of categories and \( \lfloor . \rfloor \) is the function that takes the greatest integer that is no larger than the argument of the function. If the cutpoints are specified by the user, the tally is made in the interval to which \( x_i \) belongs using the endpoints specified by the user. Thus, if the computer time required to calculate the cumulative distribution function is large, user-specified cutpoints may be preferred in order to reduce the total computing time.

If the expected count in any cell is less than 1, then a rule of thumb is that the chi-squared approximation may be suspect. A warning message to this effect is issued in this case, as well as when an expected value is less than 5.
Example: The Chi-squared Goodness-of-fit Test

In this example, a discrete binomial random sample of size 1000 with binomial parameter $p = 0.3$ and binomial sample size 5 is generated via Random.nextBinomial. Random.setSeed is first used to set the seed. After the ChiSquaredTest constructor is called, the random observations are added to the test one at a time to simulate streaming data. The Chi-squared statistic, $p$-value, and Degrees of freedom are then computed and printed.

```csharp
using System;
using Imsl.Stat;

class ChiSquaredTestEx1 : ICdfFunction
{
    public double CdfFunction(double x)
    {
        return Cdf.Binomial((int) x, 5, 0.3);
    }

    public static void Main(String[] args)
    {
        // Seed the random number generator
        rn.Multiplier = 16807;

        // Construct a ChiSquaredTest object
        ICdfFunction bindf = new ChiSquaredTestEx1();
        double[] cutp = new double[] { 0.5, 1.5, 2.5, 3.5, 4.5 };
        int nParameters = 0;
        ChiSquaredTest cst = new ChiSquaredTest(bindf, cutp, nParameters);
        for (int i = 0; i < 1000; i++)
        {
            cst.Update(rn.NextBinomial(5, 0.3), 1.0);
        }

        // Print goodness-of-fit test statistics
        Console.Out.WriteLine("The Chi-squared statistic is " + cst.ChiSquared);
        Console.Out.WriteLine("The P-value is " + cst.P);
        Console.Out.WriteLine("The Degrees of freedom are " + cst.DegreesOfFreedom);
    }
}
```

Output

The Chi-squared statistic is 4.79629666357385
The P-value is 0.441242957205531
The Degrees of freedom are 5
Imsl.Stat.ChiSquaredTest: An expected value is less than five.
NormalityTest Class

Performs a test for normality.

public class Imsl.Stat.NormalityTest

Properties

ChiSquared
  public double ChiSquared {get; }
  Returns the chi-square statistic for the chi-squared goodness-of-fit test. A double scalar
  containing the chi-square statistic.
  Returns Double.NaN for other tests.

DegreesOfFreedom
  public double DegreesOfFreedom {get; }
  Returns the degrees of freedom for the chi-squared goodness-of-fit test. A double scalar
  containing the degrees of freedom.
  Returns Double.NaN for other tests.

MaxDifference
  public double MaxDifference {get; }
  Returns the maximum absolute difference between the empirical and the theoretical
  distributions for the Lilliefors test. A double scalar containing the maximum absolute
  difference between the empirical and the theoretical distributions.
  Returns Double.NaN for other tests.

ShapiroWilkW
  public double ShapiroWilkW {get; }
  Returns the Shapiro-Wilk W statistic for the Shapiro-Wilk W test. A double scalar
  containing the Shapiro-Wilk W statistic.
  Returns Double.NaN for other tests.

Constructor

NormalityTest
  public NormalityTest(double[] x)
  Constructor for NormalityTest.
  x.length must be in the range from 3 to 2,000, inclusive, for the Shapiro-Wilk W test
  and must be greater than 4 for the Lilliefors test.
  x  A double array containing the observations.
Methods

ChiSquaredTest

```java
public double ChiSquaredTest(int n)
Performs the chi-squared goodness-of-fit test.

```

**n** A `int` scalar containing the number of cells into which the observations are to be tallied.

Returns — A `double` scalar containing the p-value for the chi-squared goodness-of-fit test.

**Imsl.Stat.NoVariationInputException** is thrown if there is no variation in the input data

**Imsl.Stat.DidNotConvergeException** is thrown if the iteration did not converge

LillieforsTest

```java
public double LillieforsTest()
Performs the Lilliefors test.

Probabilities less than 0.01 are reported as 0.01, and probabilities greater than 0.10 for the normal distribution are reported as 0.5. Otherwise, an approximate probability is computed.

Returns — A `double` scalar containing the p-value for the Lilliefors test.

**Imsl.Stat.NoVariationInputException** is thrown if there is no variation in the input data

**Imsl.Stat.DidNotConvergeException** is thrown if the iteration did not converge

ShapiroWilkWTest

```java
public double ShapiroWilkWTest()
Performs the Shapiro-Wilk W test.

Returns — A `double` scalar containing the p-value for the Shapiro-Wilk W test.

**Imsl.Stat.NoVariationInputException** is thrown if there is no variation in the input data

**Imsl.Stat.DidNotConvergeException** is thrown if the iteration did not converge

Description

Three methods are provided for testing normality: the Shapiro-Wilk W test, the Lilliefors test, and the chi-squared test.

Shapiro-Wilk W Test

The Shapiro-Wilk W test is thought by D’Agostino and Stevens (1986, p. 406) to be one of the best omnibus tests of normality. The function is based on the approximations and code given
by Royston (1982a, b, c). It can be used in samples as large as 2,000 or as small as 3. In the
Shapiro and Wilk test, \( W \) is given by

\[
W = \left( \sum a_i x_{(i)} \right)^2 / \left( \sum (x_i - \bar{x})^2 \right)
\]

where \( x_{(i)} \) is the \( i \)-th largest order statistic and \( \bar{x} \) is the sample mean. Royston (1982) gives
approximations and tabled values that can be used to compute the coefficients \( a_i, i = 1, \ldots, n \),
and obtains the significance level of the \( W \) statistic.

**Lilliefors Test**

This function computes Lilliefors test and its \( p \)-values for a normal distribution in which both
the mean and variance are estimated. The one-sample, two-sided Kolmogorov-Smirnov statistic
\( D \) is first computed. The \( p \)-values are then computed using an analytic approximation given by
Dallal and Wilkinson (1986). Because Dallal and Wilkinson give approximations in the range
\((0.01, 0.10)\) if the computed probability of a greater \( D \) is less than 0.01, the \( p \)-value is set to
0.50. Note that because parameters are estimated, \( p \)-values in Lilliefors test are not the same as
in the Kolmogorov-Smirnov Test.

Observations should not be tied. If tied observations are found, an informational message is
printed. A general reference for the Lilliefors test is Conover (1980). The original reference for
the test for normality is Lilliefors (1967).

**Chi-Squared Test**

This function computes the chi-squared statistic, its \( p \)-value, and the degrees of freedom of the
test. Argument \( n \) finds the number of intervals into which the observations are to be divided.
The intervals are equiprobable except for the first and last interval, which are infinite in length.

If more flexibility is desired for the specification of intervals, the same test can be performed
with class \texttt{ChiSquaredTest}.

**Example: Shapiro-Wilk W Test**

The following example is taken from Conover (1980, pp. 195, 364). The data consists of 50
two-digit numbers taken from a telephone book. The \( W \) test fails to reject the null hypothesis
of normality at the .05 level of significance.

```csharp
using System;
using Imsl;
using Imsl.Stat;

class NormalityTestEx1
{
    public static void Main(String[] args)
    {
        double[] x = new double[] { 23.0, 36.0, 54.0, 61.0, 73.0, 23.0, 37.0, 54.0, 61.0, 73.0, 24.0, 40.0, ...
```
NormalityTest nt = new NormalityTest(x);

Console.Out.WriteLine
  ("p-value = " + nt.ShapiroWilkWTest().ToString("0.0000");
Console.Out.WriteLine("Shapiro Wilk W Statistic = " +
  nt.ShapiroWilkW.ToString("0.0000");
}

Output

p-value = 0.2309
Shapiro Wilk W Statistic = 0.9642
Chapter 18: Time Series and Forecasting

Types

class AutoCorrelation ................................................................. 415
enumeration AutoCorrelation.StdErr ............................................... 423
class CrossCorrelation ................................................................. 424
enumeration CrossCorrelation.StdErr ............................................. 435
class MultiCrossCorrelation ......................................................... 435
class ARMA ................................................................. 446
enumeration ARMA.ParamEstimation .............................................. 464
class Difference ................................................................. 464
class GARCH ................................................................. 469
class KalmanFilter ................................................................. 474

Usage Notes

The classes in this chapter assume the time series does not contain any missing observations. If missing values are present, they should be set to NaN (see Double.NaN), and the classes will return an appropriate error message. To enable fitting of the model, the missing values must be replaced by appropriate estimates.

General Methodology

A major component of the model identification step concerns determining if a given time series is stationary. The sample correlation functions computed by the AutoCorrelation class methods getAutoCorrelations and getPartialAutoCorrelations may be used to diagnose the presence of nonstationarity in the data, as well as to indicate the type of transformation required to induce stationarity.

The "raw" data and sample correlation functions provide insight into the nature of the
underlying model. Typically, this information is displayed in graphical form via time series plots, plots of the lagged data, and various correlation function plots.

**ARIMA Model (Autoregressive Integrated Moving Average)**

A small, yet comprehensive, class of stationary time-series models consists of the nonseasonal ARMA processes defined by

\[ \phi(B) (W_t - \mu) = \theta(B) A_t, \quad t \in \mathbb{Z} \]

where \( Z = \ldots, -2, -1, 0, 1, 2, \ldots \) denotes the set of integers, \( B \) is the backward shift operator defined by \( B^k W_t = W_{t-k} \), \( \mu \) is the mean of \( W_t \), and the following equations are true:

\[ \phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \cdots - \phi_p B^p, p \geq 0 \]

\[ \theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \cdots - \theta_q B^q, q \geq 0 \]

The model is of order \((p, q)\) and is referred to as an ARMA \((p, q)\) model.

An equivalent version of the ARMA \((p, q)\) model is given by

\[ \phi(B) W_t = \theta_0 + \theta(B) A_t, \quad t \in \mathbb{Z} \]

where \( \theta_0 \) is an overall constant defined by the following:

\[ \theta_0 = \mu \left( 1 - \sum_{i=1}^{p} \phi_i \right) \]

See Box and Jenkins (1976, pp. 92-93) for a discussion of the meaning and usefulness of the overall constant.

If the “raw” data, \( \{ Z_t \} \), are homogeneous and nonstationary, then differencing using the Difference class induces stationarity, and the model is called ARIMA (AutoRegressive Integrated Moving Average). Parameter estimation is performed on the stationary time series \( W_t = \Delta^d Z_t \), where \( \Delta^d = (1 - B)^d \) is the backward difference operator with period 1 and order \( d, d > 0 \).

Typically, the method of moments includes setting property \texttt{Method} to \texttt{MethodOfMoments} in the ARMA class for preliminary parameter estimates. These estimates can be used as initial values into the least-squares procedure by setting property \texttt{Method} to \texttt{LeastSquares} in the ARMA class. Other initial estimates provided by the user can be used. The least-squares procedure can be used to compute conditional or unconditional least-squares estimates of the parameters, depending on the choice of the backcasting length. The parameter estimates from either the method of moments or least-squares procedures can be used in the \texttt{forecast} method. The
functions for preliminary parameter estimation, least-squares parameter estimation, and forecasting follow the approach of Box and Jenkins (1976, Programs 2-4, pp. 498-509).

**AutoCorrelation Class**

Computes the sample autocorrelation function of a stationary time series.

```java
public class Imsl.Stat.AutoCorrelation
```

**Properties**

Mean

```java
public double Mean {get; set; }
```

The mean of the time series `x`. A `double` containing the mean of the time series `x`.

Variance

```java
public double Variance {get; }
```

Returns the variance of the time series `x`. A `double` containing the variance of the time series `x`.

**Constructor**

```java
AutoCorrelation
```

```java
public AutoCorrelation(double[] x, int maximumLag)
```

Constructor to compute the sample autocorrelation function of a stationary time series.

- `maximumLag` must be greater than or equal to 1 and less than the number of observations in `x`.

- `x` A one-dimensional `double` array containing the stationary time series.

- `maximumLag` An `int` containing the maximum lag of autocovariance, autocorrelations, and standard errors of autocorrelations to be computed.

**Methods**

GetAutoCorrelations

```java
public double[] GetAutoCorrelations()
```

Returns the autocorrelations of the time series `x`.

- The 0-th element of this array is 1. The `k`-th element of this array contains the autocorrelation of lag `k` where `k = 1, ..., maximumLag`. 
Returns — A double array of length maximumLag +1 containing the autocorrelations of the time series x.

GetAutoCovariances
public double[] GetAutoCovariances()
Returns the variance and autocovariances of the time series x.

The θ-th element of the array contains the variance of the time series x. The k-th element contains the autocovariance of lag k where k = 1, ..., maximumLag.

Returns — A double array of length maximumLag + 1 containing the variances and autocovariances of the time series x.

Imsl.Stat.NonPosVarianceException is thrown if the problem is ill-conditioned.

GetPartialAutoCorrelations
public double[] GetPartialAutoCorrelations()
Returns the sample partial autocorrelation function of the stationary time series x.

Returns — A double array of length maximumLag containing the partial autocorrelations of the time series x.

GetStandardErrors
Returns the standard errors of the autocorrelations of the time series x.

Method of computation for standard errors of the autocorrelation is chosen by the stderrMethod parameter.

If stderrMethod is set to Bartletts, Bartlett’s formula is used to compute the standard errors of autocorrelations.

If stderrMethod is set to Morans, Moran’s formula is used to compute the standard errors of autocorrelations.

stderrMethod An int specifying the method to compute the standard errors of autocorrelations of the time series x.

Returns — A double array of length maximumLag containing the standard errors of the autocorrelations of the time series x.

Description

AutoCorrelation estimates the autocorrelation function of a stationary time series given a sample of n observations \( \{X_t\} \) for t = 1, 2, ..., n.

Let

\[ \hat{\mu} = \bar{x} \]

be the estimate of the mean \( \mu \) of the time series \( \{X_t\} \) where
\[ \hat{\mu} = \begin{cases} \mu & \text{for } \mu \text{ known} \\ \frac{1}{n} \sum_{t=1}^{n} X_t & \text{for } \mu \text{ unknown} \end{cases} \]

The autocovariance function \( \sigma(k) \) is estimated by

\[ \hat{\sigma}(k) = \frac{1}{n} \sum_{t=1}^{n-k} (X_t - \hat{\mu}) (X_{t+k} - \hat{\mu}), \quad k=0,1,\ldots,K \]

where \( K = \text{maximumLag} \). Note that \( \hat{\sigma}(0) \) is an estimate of the sample variance. The autocorrelation function \( \rho(k) \) is estimated by

\[ \hat{\rho}(k) = \frac{\hat{\sigma}(k)}{\hat{\sigma}(0)}, \quad k = 0,1,\ldots,K \]

Note that \( \hat{\rho}(0) \equiv 1 \) by definition.

The standard errors of sample autocorrelations may be optionally computed according to the \textit{GetStandardErrors} method argument \texttt{stderrMethod}. One method (Bartlett 1946) is based on a general asymptotic expression for the variance of the sample autocorrelation coefficient of a stationary time series with independent, identically distributed normal errors. The theoretical formula is

\[ \text{var}\{\hat{\rho}(k)\} = \frac{1}{n} \sum_{i=-\infty}^{\infty} \left[ \rho^2(i) + \rho(i-k)\rho(i+k) - 4\rho(i)\rho(k)\rho(i-k) + 2\rho^2(i)\rho^2(k) \right] \]

where \( \hat{\rho}(k) \) assumes \( \mu \) is unknown. For computational purposes, the autocorrelations \( \rho(k) \) are replaced by their estimates \( \hat{\rho}(k) \) for \( |k| \leq K \), and the limits of summation are bounded because of the assumption that \( \rho(k) = 0 \) for all \( k \) such that \( |k| > K \).

A second method (Moran 1947) utilizes an exact formula for the variance of the sample autocorrelation coefficient of a random process with independent, identically distributed normal errors. The theoretical formula is

\[ \text{var}\{\hat{\rho}(k)\} = \frac{n-k}{n(n+2)} \]

where \( \mu \) is assumed to be equal to zero. Note that this formula does not depend on the autocorrelation function.

The method \textit{GetPartialAutoCorrelations} returns the estimated partial autocorrelations of the stationary time series given \( K = \text{maximumLag} \) sample autocorrelations \( \hat{\rho}(k) \) for \( k=0,1,\ldots,K \).

Consider the AR(\( k \)) process defined by

\[ X_t = \phi_{k1}X_{t-1} + \phi_{k2}X_{t-2} + \cdots + \phi_{kk}X_{t-k} + A_t \]
where $\phi_{kj}$ denotes the $j$-th coefficient in the process. The set of estimates $\{\hat{\phi}_{kk}\}$ for $k = 1, ..., K$ is the sample partial autocorrelation function. The autoregressive parameters $\{\hat{\phi}_{kj}\}$ for $j = 1, ..., k$ are approximated by Yule-Walker estimates for successive AR($k$) models where $k = 1, ..., K$. Based on the sample Yule-Walker equations

$$\hat{\rho}(j) = \hat{\phi}_{k1}\hat{\rho}(j - 1) + \hat{\phi}_{k2}\hat{\rho}(j - 2) + \cdots + \hat{\phi}_{kk}\hat{\rho}(j - k), \quad j = 1, 2, \ldots, k$$

a recursive relationship for $k = 1, ..., K$ was developed by Durbin (1960). The equations are given by

$$\hat{\phi}_{kk} = \begin{cases} \hat{\rho}(1) & \text{for } k = 1 \\ \frac{\hat{\rho}(k) - \sum_{i=1}^{k-1} \hat{\phi}_{k-1,i}\hat{\rho}(i)}{1 - \sum_{i=1}^{k-1} \hat{\phi}_{k-1,i}\hat{\rho}(i)} & \text{for } k = 2, \ldots, K \end{cases}$$

and

$$\hat{\phi}_{kj} = \begin{cases} \hat{\phi}_{k-1,j} - \hat{\phi}_{kk}\hat{\phi}_{k-1,k-j} & \text{for } j = 1, 2, \ldots, k - 1 \\ \hat{\phi}_{kk} & \text{for } j = k \end{cases}$$

This procedure is sensitive to rounding error and should not be used if the parameters are near the nonstationarity boundary. A possible alternative would be to estimate $\{\phi_{kk}\}$ for successive AR($k$) models using least or maximum likelihood. Based on the hypothesis that the true process is AR($p$), Box and Jenkins (1976, page 65) note

$$\text{var}\{\hat{\phi}_{kk}\} \approx \frac{1}{n} \quad k \geq p + 1$$

See Box and Jenkins (1976, pages 82-84) for more information concerning the partial autocorrelation function.

**Example 1: AutoCorrelation**

Consider the Wolfer Sunspot Data (Anderson 1971, p. 660) consisting of the number of sunspots observed each year from 1749 through 1924. The data set for this example consists of the number of sunspots observed from 1770 through 1869. This example computes the estimated autocovariances, estimated autocorrelations, and estimated standard errors of the autocorrelations using both Bartlett and Moran formulas.
double[] x = new double[] { 100.8, 81.6, 66.5, 34.8, 30.6, 7, 19.8, 92.5, 154.4, 125.9, 84.8, 68.1, 38.5, 22.8, 10.2, 24.1, 82.9, 132, 130.9, 118.1, 89.9, 66.6, 60, 46.9, 41, 21.3, 16, 6.4, 4.1, 6.8, 14.5, 34, 45, 43.1, 47.5, 42.2, 28.1, 10.1, 8.1, 2.5, 0, 1.4, 5, 12.2, 13.9, 35.4, 45.8, 41.1, 30.4, 23.9, 15.7, 6.6, 4, 1.8, 8.5, 16.6, 36.3, 49.7, 62.5, 67, 71, 47.8, 27.5, 8.5, 13.2, 56.9, 121.5, 138.3, 103.2, 85.8, 63.2, 36.8, 24.2, 10.7, 15, 40.1, 61.5, 98.5, 124.3, 95.9, 66.5, 64.5, 54.2, 39, 20.6, 6.7, 4.3, 22.8, 54.8, 93.8, 95.7, 77.2, 59.1, 44, 47, 30.5, 16.3, 7.3, 37.3, 73.9};

AutoCorrelation ac = new AutoCorrelation(x, 20);

new PrintMatrix("AutoCovariances are: ").Print(ac.GetAutoCovariances());
Console.Out.WriteLine();
new PrintMatrix("AutoCorrelations are: ").Print(ac.GetAutoCorrelations());
Console.Out.WriteLine("Mean = "+ac.Mean);
new PrintMatrix("Standard Error using Bartlett are: ").Print(ac.GetStandardErrors(AutoCorrelation.StdErr.Bartletts));
Console.Out.WriteLine();
new PrintMatrix("Standard Error using Moran are: ").Print(ac.GetStandardErrors(AutoCorrelation.StdErr.Morans));
Console.Out.WriteLine();
new PrintMatrix("Partial AutoCovariances: ").Print(ac.GetPartialAutoCorrelations());
ac.Mean = 50;
new PrintMatrix("AutoCovariances are: ").Print(ac.GetAutoCovariances());
Console.Out.WriteLine();
new PrintMatrix("AutoCorrelations are: ").Print(ac.GetAutoCorrelations());
new PrintMatrix("Standard Error using Bartlett are: ").Print(ac.GetStandardErrors(AutoCorrelation.StdErr.Bartletts));
}

Time Series and Forecasting

AutoCorrelation Class • 419
Output

AutoCovariances are:

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1115.02915024</td>
</tr>
<tr>
<td>2</td>
<td>592.00446848</td>
</tr>
<tr>
<td>3</td>
<td>95.29741072</td>
</tr>
<tr>
<td>4</td>
<td>-235.95179904</td>
</tr>
<tr>
<td>5</td>
<td>-370.0108088</td>
</tr>
<tr>
<td>6</td>
<td>-294.25541456</td>
</tr>
<tr>
<td>7</td>
<td>-60.44237232</td>
</tr>
<tr>
<td>8</td>
<td>227.63259792</td>
</tr>
<tr>
<td>9</td>
<td>458.38076816</td>
</tr>
<tr>
<td>10</td>
<td>567.8407384</td>
</tr>
<tr>
<td>11</td>
<td>546.12202864</td>
</tr>
<tr>
<td>12</td>
<td>398.93728688</td>
</tr>
<tr>
<td>13</td>
<td>197.75742912</td>
</tr>
<tr>
<td>14</td>
<td>26.89107936</td>
</tr>
<tr>
<td>15</td>
<td>-77.2807224</td>
</tr>
<tr>
<td>16</td>
<td>-143.73279616</td>
</tr>
<tr>
<td>17</td>
<td>-202.04799792</td>
</tr>
<tr>
<td>18</td>
<td>-245.3723168</td>
</tr>
<tr>
<td>19</td>
<td>-230.81567344</td>
</tr>
<tr>
<td>20</td>
<td>-142.8788232</td>
</tr>
</tbody>
</table>

AutoCorrelations are:

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0.806293065691258</td>
</tr>
<tr>
<td>2</td>
<td>0.428086653780237</td>
</tr>
<tr>
<td>3</td>
<td>0.0689108813212006</td>
</tr>
<tr>
<td>4</td>
<td>-0.170620023128885</td>
</tr>
<tr>
<td>5</td>
<td>-0.267559955093586</td>
</tr>
<tr>
<td>6</td>
<td>-0.212780177317129</td>
</tr>
<tr>
<td>7</td>
<td>-0.043706718936501</td>
</tr>
<tr>
<td>8</td>
<td>0.164604293249802</td>
</tr>
<tr>
<td>9</td>
<td>0.331461500117813</td>
</tr>
<tr>
<td>10</td>
<td>0.410613524938228</td>
</tr>
<tr>
<td>11</td>
<td>0.394908424249623</td>
</tr>
<tr>
<td>12</td>
<td>0.288477093166393</td>
</tr>
<tr>
<td>13</td>
<td>0.143001143740562</td>
</tr>
<tr>
<td>14</td>
<td>0.0194453129877855</td>
</tr>
<tr>
<td>15</td>
<td>-0.0558827637549379</td>
</tr>
<tr>
<td>16</td>
<td>-0.10395181274212</td>
</tr>
<tr>
<td>17</td>
<td>-0.146103713633525</td>
</tr>
<tr>
<td>18</td>
<td>-0.17743206881559</td>
</tr>
<tr>
<td>19</td>
<td>-0.166960619369514</td>
</tr>
<tr>
<td>20</td>
<td>-0.103317661566611</td>
</tr>
</tbody>
</table>

Mean = 46.976

Standard Error using Bartlett are:

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Standard Error using Moran are:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0347838253702384</td>
</tr>
<tr>
<td>1</td>
<td>0.0962419914340011</td>
</tr>
<tr>
<td>2</td>
<td>0.156783378574532</td>
</tr>
<tr>
<td>3</td>
<td>0.205766777086907</td>
</tr>
<tr>
<td>4</td>
<td>0.230956575779118</td>
</tr>
<tr>
<td>5</td>
<td>0.22894712235613</td>
</tr>
<tr>
<td>6</td>
<td>0.208621905639667</td>
</tr>
<tr>
<td>7</td>
<td>0.178475936561125</td>
</tr>
<tr>
<td>8</td>
<td>0.145727084432033</td>
</tr>
<tr>
<td>9</td>
<td>0.134405581638002</td>
</tr>
<tr>
<td>10</td>
<td>0.150675803916788</td>
</tr>
<tr>
<td>11</td>
<td>0.174348147103935</td>
</tr>
<tr>
<td>12</td>
<td>0.19061947429408</td>
</tr>
<tr>
<td>13</td>
<td>0.195490061669564</td>
</tr>
<tr>
<td>14</td>
<td>0.195892530944597</td>
</tr>
<tr>
<td>15</td>
<td>0.198285328179468</td>
</tr>
<tr>
<td>16</td>
<td>0.196020624500033</td>
</tr>
<tr>
<td>17</td>
<td>0.198716030900604</td>
</tr>
<tr>
<td>18</td>
<td>0.205358590947539</td>
</tr>
<tr>
<td>19</td>
<td>0.2093988822353</td>
</tr>
</tbody>
</table>

Partial AutoCovariances:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-0.634544877310468</td>
</tr>
<tr>
<td>1</td>
<td>-0.058660846682815</td>
</tr>
<tr>
<td>2</td>
<td>-0.00094221571933657</td>
</tr>
<tr>
<td>3</td>
<td>0.171719898229681</td>
</tr>
<tr>
<td>4</td>
<td>0.108591873581717</td>
</tr>
<tr>
<td>5</td>
<td>0.11000138764865</td>
</tr>
</tbody>
</table>

Time Series and Forecasting AutoCorrelation Class • 421
AutoCovariances are:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1392.0526</td>
</tr>
<tr>
<td>1</td>
<td>1126.5241</td>
</tr>
<tr>
<td>2</td>
<td>604.1624</td>
</tr>
<tr>
<td>3</td>
<td>106.7545</td>
</tr>
<tr>
<td>4</td>
<td>-225.882</td>
</tr>
<tr>
<td>5</td>
<td>-361.0259</td>
</tr>
<tr>
<td>6</td>
<td>-286.5701</td>
</tr>
<tr>
<td>7</td>
<td>-53.7603</td>
</tr>
<tr>
<td>8</td>
<td>235.9665</td>
</tr>
<tr>
<td>9</td>
<td>470.7857</td>
</tr>
<tr>
<td>10</td>
<td>584.0143</td>
</tr>
<tr>
<td>11</td>
<td>564.7639</td>
</tr>
<tr>
<td>12</td>
<td>418.3631</td>
</tr>
<tr>
<td>13</td>
<td>216.1044</td>
</tr>
<tr>
<td>14</td>
<td>43.125</td>
</tr>
<tr>
<td>15</td>
<td>-63.4683</td>
</tr>
<tr>
<td>16</td>
<td>-131.5012</td>
</tr>
<tr>
<td>17</td>
<td>-189.0627</td>
</tr>
<tr>
<td>18</td>
<td>-229.6888</td>
</tr>
<tr>
<td>19</td>
<td>-212.1559</td>
</tr>
<tr>
<td>20</td>
<td>-121.5693</td>
</tr>
</tbody>
</table>

AutoCorrelations are:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0.809253975029392</td>
</tr>
<tr>
<td>2</td>
<td>0.434008312616923</td>
</tr>
<tr>
<td>3</td>
<td>0.076688532917362</td>
</tr>
<tr>
<td>4</td>
<td>-0.162265420142888</td>
</tr>
<tr>
<td>5</td>
<td>-0.259347886710603</td>
</tr>
<tr>
<td>6</td>
<td>-0.205861545749061</td>
</tr>
<tr>
<td>7</td>
<td>-0.0386194458456527</td>
</tr>
<tr>
<td>8</td>
<td>0.16950975846746</td>
</tr>
<tr>
<td>9</td>
<td>0.338195338308337</td>
</tr>
<tr>
<td>10</td>
<td>0.419534649768263</td>
</tr>
<tr>
<td>11</td>
<td>0.405705861976767</td>
</tr>
<tr>
<td>12</td>
<td>0.300536847530043</td>
</tr>
<tr>
<td>13</td>
<td>0.155241547625427</td>
</tr>
<tr>
<td>14</td>
<td>0.0309794328174093</td>
</tr>
<tr>
<td>15</td>
<td>-0.04559332025241</td>
</tr>
</tbody>
</table>
Standard Error using Bartlett are:

```
  0
  1  0.0344591054641365
  2  0.0972222809088609
  3  0.1594741003303087
  4  0.209799660647689
  5  0.235599778243579
  6  0.233236443705991
  7  0.211657508693761
  8  0.180412936841618
  9  0.146896536063348
10  0.133747601649498
11  0.148150190923942
12  0.1722823511000035
13  0.190275929042947
14  0.196791614240352
15  0.197983743593071
16  0.19847478794747
17  0.198318159677368
18  0.201022833791806
19  0.207071652966429
20  0.210217650328868
```

**AutoCorrelation.StdErr Enumeration**

Standard Error computation method.

```java
public enumeration Imsl.Stat.AutoCorrelation.StdErr
```

**Fields**

- **Bartletts**
  ```java
  public Imsl.Stat.AutoCorrelation.StdErr Bartletts
  Indicates standard error computation using Bartlett’s formula.
  ```

- **Morans**
  ```java
  public Imsl.Stat.AutoCorrelation.StdErr Morans
  Indicates standard error computation using Moran’s formula.
  ```
CrossCorrelation Class

Computes the sample cross-correlation function of two stationary time series.

```csharp
public class Imsl.Stat.CrossCorrelation
{

Properties

MeanX
public double MeanX {get; set; }
Estimate of the mean of time series \( x \). A `double` containing the estimate of the mean of time series \( x \).

MeanY
public double MeanY {get; set; }
Estimate of the mean of time series \( y \). A `double` containing the estimate of the mean of time series \( y \).

VarianceX
public double VarianceX {get; }
Returns the variance of time series \( x \). A `double` containing the variance of the time series \( x \).

VarianceY
public double VarianceY {get; }
Returns the variance of time series \( y \). A `double` containing the variance of the time series \( y \).

Constructor

CrossCorrelation
public CrossCorrelation(double[] x, double[] y, int maximumLag)
Constructor to compute the sample cross-correlation function of two stationary time series.

maximumLag must be greater than or equal to 1 and less than the minimum of the number of observations of \( x \) and \( y \).

\( x \) A one-dimensional `double` array containing the first stationary time series.
\( y \) A one-dimensional `double` array containing the second stationary time series.
maximumLag An `int` containing the maximum lag of the cross-covariance and cross-correlations to be computed.
}
Methods

GetAutoCorrelationX
public double[] GetAutoCorrelationX()
Returns the autocorrelations of the time series x.

The \( \theta \)-th element of this array is 1. The \( k \)-th element of this array contains the
autocorrelation of lag \( k \) where \( k = 1, \ldots, \text{maximumLag} \).

Returns — A double array of length \( \text{maximumLag} + 1 \) containing the autocorrelations of
the time series x.

Imsl.Stat.NonPosVarianceException is thrown if the problem is ill-conditioned.

GetAutoCorrelationY
public double[] GetAutoCorrelationY()
Returns the autocorrelations of the time series y.

The \( \theta \)-th element of this array is 1. The \( k \)-th element of this array contains the
autocorrelation of lag \( k \) where \( k = 1, \ldots, \text{maximumLag} \).

Returns — A double array of length \( \text{maximumLag} + 1 \) containing the autocorrelations of
the time series y.

Imsl.Stat.NonPosVarianceException is thrown if the problem is ill-conditioned.

GetAutoCovarianceX
public double[] GetAutoCovarianceX()
Returns the autocovariances of the time series x.

The \( \theta \)-th element of the array contains the variance of the time series x. The \( k \)-th
elements contains the autocovariance of lag \( k \) where \( k = 1, \ldots, \text{maximumLag} \).

Returns — A double array of length \( \text{maximumLag} + 1 \) containing the variances and
autocovariances of the time series x.

Imsl.Stat.NonPosVarianceException is thrown if the problem is ill-conditioned.

GetAutoCovarianceY
public double[] GetAutoCovarianceY()
Returns the autocovariances of the time series y.

The \( \theta \)-th element of the array contains the variance of the time series y. The \( k \)-th
elements contains the autocovariance of lag \( k \) where \( k = 1, \ldots, \text{maximumLag} \).

Returns — A double array of length \( \text{maximumLag} + 1 \) containing the variances and
autocovariances of the time series y.

Imsl.Stat.NonPosVarianceException is thrown if the problem is ill-conditioned.

GetCrossCorrelations
public double[] GetCrossCorrelations()
Returns the cross-correlations between the time series x and y.
The cross-correlation between \(x\) and \(y\) at lag \(k\), where \(k = -\text{maximumLag},..., 0, 1,\ldots, \text{maximumLag}\), corresponds to output array indices 0, 1,\ldots, \((2\times\text{maximumLag})\).

Returns — A \(\text{double}\) array of length \(2 \times \text{maximumLag} + 1\) containing the cross-correlations between the time series \(x\) and \(y\).

\text{Imsl.Stat.NonPosVarianceXYException} is thrown if the problem is ill-conditioned. The variance is too small to work with.

\text{GetCrossCovariances}

\text{public double[]} \text{GetCrossCovariances()}

Returns the cross-covariances between the time series \(x\) and \(y\).

The cross-covariance between \(x\) and \(y\) at lag \(k\), where \(k = -\text{maximumLag},..., 0, 1,\ldots, \text{maximumLag}\), corresponds to output array indices 0, 1,\ldots, \((2\times\text{maximumLag})\).

Returns — A \(\text{double}\) array of length \(2 \times \text{maximumLag} + 1\) containing the cross-covariances between the time series \(x\) and \(y\).

\text{GetStandardErrors}

\text{public double[]} \text{GetStandardErrors(Imsl.Stat.CrossCorrelation.StdErr stderrMethod)}

Returns the standard errors of the cross-correlations between the time series \(x\) and \(y\).

The standard error of cross-correlations between \(x\) and \(y\) at lag \(k\), where \(k = -\text{maximumLag},..., 0, 1,\ldots, \text{maximumLag}\), corresponds to output array indices 0, 1,\ldots, \((2\times\text{maximumLag})\).

Method of computation for standard errors of the cross-correlation is determined by the \text{stderrMethod} parameter. If \text{stderrMethod} is set to \text{Bartletts}, Bartlett’s formula is used to compute the standard errors of cross-correlations. If \text{stderrMethod} is set to \text{BartlettsNoCC}, Bartlett’s formula is used to compute the standard errors of cross-correlations, with the assumption of no cross-correlation.

\text{stderrMethod} An \text{int} specifying the method to compute the standard errors of cross-correlations between the time series \(x\) and \(y\).

Returns — A \(\text{double}\) array of length \(2 \times \text{maximumLag} + 1\) containing the standard errors of the cross-correlations between the time series \(x\) and \(y\).

\text{Imsl.Stat.NonPosVarianceException} is thrown if the problem is ill-conditioned.

\text{Description}

\text{CrossCorrelation} estimates the cross-correlation function of two jointly stationary time series given a sample of \(n = x.\text{Length}\) observations \(\{X_t\}\) and \(\{Y_t\}\) for \(t = 1,2,\ldots, n\).

Let

\[
\hat{\mu}_x = x\text{mean}
\]
be the estimate of the mean \( \mu_X \) of the time series \( \{X_t\} \) where
\[
\hat{\mu}_X = \begin{cases} 
\mu_X & \text{for } \mu_X \text{ known} \\
\frac{1}{n} \sum_{t=1}^{n} X_t & \text{for } \mu_X \text{ unknown}
\end{cases}
\]

The autocovariance function of \( \{X_t\} \), \( \sigma_X(k) \), is estimated by
\[
\hat{\sigma}_X(k) = \frac{1}{n} \sum_{t=1}^{n-k} (X_t - \hat{\mu}_X)(X_{t+k} - \hat{\mu}_X), \quad k=0,1,\ldots,K
\]
where \( K = \text{maximumLag} \). Note that \( \hat{\sigma}_X(0) \) is equivalent to the sample variance of \( x \) returned by property \text{VarianceX}. The autocorrelation function \( \rho_X(k) \) is estimated by
\[
\hat{\rho}_X(k) = \frac{\hat{\sigma}_X(k)}{\hat{\sigma}_X(0)}, \quad k = 0, 1, \ldots, K
\]
Note that \( \hat{\rho}_x(0) \equiv 1 \) by definition. Let
\[
\hat{\mu}_Y = \text{ymean}, \hat{\sigma}_Y(k), \text{and } \hat{\rho}_Y(k)
\]
be similarly defined.

The cross-covariance function \( \sigma_{XY}(k) \) is estimated by
\[
\hat{\sigma}_{XY}(k) = \begin{cases} 
\frac{1}{n} \sum_{t=1}^{n-k} (X_t - \hat{\mu}_X)(Y_{t+k} - \hat{\mu}_Y) & k = 0, 1, \ldots, K \\
\frac{1}{n} \sum_{t=1-k}^{n} (X_t - \hat{\mu}_X)(Y_{t+k} - \hat{\mu}_Y) & k = -1, -2, \ldots, -K
\end{cases}
\]
The cross-correlation function \( \rho_{XY}(k) \) is estimated by
\[
\hat{\rho}_{XY}(k) = \frac{\hat{\sigma}_{XY}(k)}{|\hat{\sigma}_X(0)\hat{\sigma}_Y(0)|^\frac{1}{2}} \quad k = 0, \pm 1, \ldots, \pm K
\]
The standard errors of the sample cross-correlations may be optionally computed according to the \text{GetStandardErrors} method argument \text{stderrMethod}. One method is based on a general asymptotic expression for the variance of the sample cross-correlation coefficient of two jointly stationary time series with independent, identically distributed normal errors given by Bartlett (1978, page 352). The theoretical formula is
\[
\text{var} \{ \hat{\rho}_{XY}(k) \} = \frac{1}{n-k} \sum_{i=-\infty}^{\infty} \left[ \rho_X(i) + \rho_{XY}(i-k)\rho_{XY}(i+k) \\
-2\rho_{XY}(k)\{\rho_X(i)\rho_{XY}(i+k) + \rho_{XY}(-i)\rho_Y(i+k)\} \\
+ \rho_{XY}^2(k)\{\rho_X(i) + \frac{1}{2}\rho_X^2(i) + \frac{1}{2}\rho_Y^2(i)\} \right]
\]
For computational purposes, the autocorrelations \( \rho_X(k) \) and \( \rho_Y(k) \) and the cross-correlations \( \rho_{XY}(k) \) are replaced by their corresponding estimates for \( |k| \leq K \), and the limits of summation are equal to zero for all \( k \) such that \( |k| > K \).
A second method evaluates Bartlett’s formula under the additional assumption that the two series have no cross-correlation. The theoretical formula is

$$\text{var}\{\hat{\rho}_{XY}(k)\} = \frac{1}{n-k} \sum_{i=-\infty}^{\infty} \rho_X(i)\rho_Y(i) \quad k \geq 0$$

For additional special cases of Bartlett’s formula, see Box and Jenkins (1976, page 377).

An important property of the cross-covariance coefficient is $\sigma_{XY}(k) = \sigma_{YX}(-k)$ for $k \geq 0$. This result is used in the computation of the standard error of the sample cross-correlation for lag $k < 0$. In general, the cross-covariance function is not symmetric about zero so both positive and negative lags are of interest.

**Example 1: CrossCorrelation**

Consider the Gas Furnace Data (Box and Jenkins 1976, pages 532-533) where $X$ is the input gas rate in cubic feet/minute and $Y$ is the percent CO$_2$ in the outlet gas. The CrossCorrelation methods GetCrossCovariance and GetCrossCorrelation are used to compute the cross-covariances and cross-correlations between time series $X$ and $Y$ with lags from $-\text{maximumLag} = -10$ through lag $\text{maximumLag} = 10$. In addition, the estimated standard errors of the estimated cross-correlations are computed. In the first invocation of method GetStandardErrors stderrMethod = Bartletts, the standard errors are based on the assumption that autocorrelations and cross-correlations for lags greater than $\text{maximumLag}$ or less than $-\text{maximumLag}$ are zero. In the second invocation of method GetStandardErrors with stderrMethod = BartlettsNoCC, the standard errors are based on the additional assumption that all cross-correlations for $X$ and $Y$ are zero.

```csharp
using System;
using Imsl.Stat;
using PrintMatrix = Imsl.Math.PrintMatrix;

class CrossCorrelationEx1
{
    public static void Main(String[] args)
    {
        double[] x2 = new double[100] { 100.8, 81.6, 66.5, 34.8, 30.6, 7, 19.8, 92.5, 154.4, 125.9, 84.8, 68.1, 38.5, 22.8, 10.2, 24.1, 82.9, 132, 130.9, 118.1, 89.9, 66.6, 60, 46.9, 41, 21.3, 16, 6.4, 4.1, 6.8, 14.5, 34, 45, 43.1, 47.5, 42.2, 28.1, 10.1, 8.1, 2.5, 0, 1.4, 5, 12.2, 13.9, 35.4, 45.8, 41.1, 30.4, 23.9, 15.7, 6.6, 4, 1.8, 8.5, 16.6, 36.3, 49.7, 62.5, 67, 71, 47.8, 27.5, 8.5, 13.2, 56.9, 121.5, 138.3, 103.2, 85.8, 63.2, 36.8, 24.2, 10.7, 15,
        ...
```
40.1, 61.5, 98.5, 124.3, 95.9,
66.5, 64.5, 54.2, 39, 20.6,
6.7, 4.3, 22.8, 54.8, 93.8,
95.7, 77.2, 59.1, 44, 47,
30.5, 16.3, 7.3, 37.3, 73.9;

double[] x = new double[]{ -0.109, 0.0, 0.178, 0.339, 0.373,
0.441, 0.461, 0.348, 0.127,
-0.18, -0.588, -1.055, -1.421,
-1.52, -1.302, -0.814, -0.475,
-0.193, 0.088, 0.435, 0.771,
0.866, 0.875, 0.891, 0.987, 1.263,
1.775, 1.976, 1.934, 1.866, 1.832,
1.767, 1.608, 1.265, 0.79, 0.36,
0.115, 0.088, 0.331, 0.645, 0.96,
1.409, 2.67, 2.834, 2.812, 2.483,
1.929, 1.485, 1.214, 1.239, 1.608,
1.905, 2.023, 1.815, 0.535, 0.122,
0.009, 0.164, 0.671, 0.1019, 0.146,
1.155, 1.112, 1.121, 1.223, 1.257,
1.157, 0.913, 0.62, 0.255, -0.28,
-1.08, -1.551, -1.799, -1.825,
-1.456, -0.944, -0.57, -0.431,
-0.577, -0.96, -1.616, -1.875,
-1.891, -1.746, -1.474, -1.201,
-0.927, -0.524, 0.04, 0.788,
0.943, 0.93, 1.006, 1.137, 1.198,
1.054, 0.595, -0.08, -0.314,
-0.288, -0.153, -0.109, -0.187,
-0.255, -0.229, -0.007, 0.254,
0.33, 0.102, -0.423, -1.139,
-2.275, -2.594, -2.716, -2.51,
-1.79, -1.346, -1.081, -0.91,
-0.876, -0.885, -0.8, -0.544,
-0.416, -0.271, 0.0, 0.403,
0.841, 1.285, 1.607, 1.746, 1.683,
1.485, 0.993, 0.648, 0.577, 0.577,
0.632, 0.747, 0.9, 0.993, 0.968,
0.79, 0.399, -0.161, -0.553,
-0.603, -0.424, -0.194, -0.049,
0.06, 0.161, 0.301, 0.517, 0.566,
0.56, 0.573, 0.592, 0.671, 0.933,
1.337, 1.46, 1.353, 0.772, 0.218,
-0.237, -0.714, -1.099, -1.269,
-1.175, -0.676, 0.033, 0.556,
0.643, 0.484, 0.109, -0.31, -0.697,
-1.047, -1.218, -1.183, -0.873,
-0.336, 0.063, 0.084, 0.0, 0.001,
0.209, 0.556, 0.782, 0.858, 0.918,
0.862, 0.416, -0.336, -0.959,
-1.813, -2.378, -2.499, -2.473,
-2.33, -2.05, -1.739, -1.261,
-0.569, -0.137, -0.024, -0.05,
-0.135, -0.276, -0.534, -0.871,
-1.243, -1.439, -1.422, -1.175,
-0.813, -0.634, -0.582, -0.625,
-0.713, -0.848, -1.039, -1.346,
- 1.628, - 1.619, - 1.149, -0.488,
- 0.16, - 0.007, - 0.092, - 0.62,
- 1.086, - 1.525, - 1.858, -2.029,
- 2.024, - 1.961, - 1.952, -1.794,
- 1.302, - 1.03, - 0.918, -0.798,
- 0.867, - 1.047, - 1.123, - 0.876,
- 0.395, 0.185, 0.662, 0.709,
0.605, 0.501, 0.603, 0.943, 1.223,
1.249, 0.824, 0.102, 0.025, 0.382,
0.922, 1.032, 0.866, 0.527, 0.093,
- 0.458, - 0.748, - 0.947, -1.029,
- 0.928, - 0.645, - 0.424, -0.276,
- 0.158, - 0.033, 0.102, 0.251,
0.28, 0.0, -0.493, -0.759, -0.824,
- 0.74, - 0.528, - 0.204, 0.034,
0.204, 0.253, 0.195, 0.131, 0.017,
- 0.182, - 0.262);

double[] y = new double[] {53.8, 53.6, 53.5, 53.4, 53.1,
52.7, 52.4, 52.2, 52.0, 52.0,
52.4, 53.0, 54.0, 54.9, 56.0,
56.8, 56.8, 56.4, 55.7, 55.0,
54.3, 53.2, 52.3, 51.6, 51.2,
50.8, 50.5, 50.0, 49.2, 48.4,
47.9, 47.6, 47.5, 47.5, 47.6,
48.1, 49.0, 50.0, 51.1, 51.8,
51.9, 51.7, 51.2, 50.0, 48.3,
47.0, 45.8, 45.6, 46.0, 46.9,
47.8, 48.2, 48.3, 47.9, 47.2,
47.2, 48.1, 49.4, 50.6, 51.5,
51.6, 51.2, 50.5, 50.1, 49.8,
49.6, 49.4, 49.3, 49.2, 49.3,
49.7, 50.3, 51.3, 52.8, 54.4,
56.0, 56.9, 57.5, 57.3, 56.6,
56.0, 55.4, 55.4, 56.4, 57.2,
58.0, 58.4, 58.4, 58.1, 57.7,
57.0, 56.0, 54.7, 53.2, 52.1,
51.6, 51.0, 50.5, 50.4, 51.0,
51.8, 52.4, 53.0, 53.4, 53.6,
53.7, 53.8, 53.8, 53.8, 53.3,
53.0, 52.9, 53.4, 54.6, 56.4,
58.0, 59.4, 60.2, 60.0, 59.4,
58.4, 57.6, 56.9, 56.4, 56.0,
55.7, 55.3, 55.0, 54.4, 53.7,
52.8, 51.6, 50.6, 49.4, 48.8,
48.5, 48.7, 49.2, 49.8, 50.4,
50.7, 50.9, 50.7, 50.5, 50.4,
50.2, 50.4, 51.2, 52.3, 53.2,
53.9, 54.1, 54.0, 53.6, 53.2,
53.0, 52.8, 52.3, 51.9, 51.6,
51.6, 51.4, 51.2, 50.7, 50.0,
49.4, 49.3, 49.7, 50.6, 51.8,
53.0, 54.0, 55.3, 55.9, 55.9,
54.6, 53.5, 52.4, 52.1, 52.3,
53.0, 53.8, 54.6, 55.4, 55.9,
55.9, 55.2, 54.4, 53.7, 53.6,
53.6, 53.2, 52.5, 52.0, 51.4,
```csharp
CrossCorrelation cc = new CrossCorrelation(x, y, 10);
Console.Out.WriteLine("Mean = "+ cc.MeanX);
Console.Out.WriteLine("Mean = "+ cc.MeanY);
Console.Out.WriteLine("Xvariance = "+ cc.VarianceX);
Console.Out.WriteLine("Yvariance = "+ cc.VarianceY);
new PrintMatrix("CrossCovariances are: ").Print(cc.GetCrossCovariances());
new PrintMatrix("CrossCorrelations are: ").Print(cc.GetCrossCorrelations());
double[] stdErrors =
cc.GetStandardErrors(CrossCorrelation.StdErr.Bartletts);
new PrintMatrix("Standard Errors using Bartlett are: ").Print(stdErrors);
stdErrors =
cc.GetStandardErrors(CrossCorrelation.StdErr.BartlettsNoCC);
new PrintMatrix("Standard Errors using Bartlett #2 are: ").Print(stdErrors);
new PrintMatrix("AutoCovariances of X are: ").Print (cc.GetAutoCovarianceX());
new PrintMatrix("AutoCovariances of Y are: ").Print (cc.GetAutoCovarianceY());
new PrintMatrix("AutoCorrelations of X are: ").Print (cc.GetAutoCorrelationX());
new PrintMatrix("AutoCorrelations of Y are: ").Print (cc.GetAutoCorrelationY());
```
Output

Mean = -0.0568344594594595
Mean = 53.5091216216216
Xvariance = 1.14693790165038
Yvariance = 10.2189370662893
CrossCovariances are:
0
0 -0.404501563294314
1 -0.508490782763824
2 -0.614369467627782
3 -0.705476130258359
4 -0.776166564117932
5 -0.831473609098764
6 -0.891315326970392
7 -0.980605209560792
8 -1.12477059434257
9 -1.34704305203341
10 -1.65862650999817
11 -2.04865124674232
12 -2.48216585776478
13 -2.88541054192018
14 -3.16536049680239
15 -3.25343758942199
16 -3.13112860301494
17 -2.83919398644463
18 -2.45302186901565
19 -2.05268794195849
20 -1.6946546517713

CrossCorrelations are:
0
0 -0.118153717307789
1 -0.148628662661878
2 -0.179455515102209
3 -0.206067503381416
4 -0.226715971265165
5 -0.242870996488244
6 -0.260350586329711
7 -0.286431898500946
8 -0.3285421835153
9 -0.393467314873008
10 -0.484450717109386
11 -0.598405005361053
12 -0.725033348897091
13 -0.842819355039272
14 -0.924592494205792
15 -0.950319553992448
16 -0.914693458680361
17 -0.829320215245049
18 -0.716520475473708
19 -0.599584112456951
20 -0.495003641096017

Standard Errors using Bartlett are:
Standard Errors using Bartlett #2 are:

0 0.15814778375455
1 0.155750271182418
2 0.152735096430409
3 0.149086745416716
4 0.145054998300008
5 0.141300099196058
6 0.138420534019813
7 0.136074039397204
8 0.132158917844376
9 0.123531347020305
10 0.107879045104545
11 0.0873410658167485
12 0.0641407975847026
13 0.0469456102701398
14 0.0440970262220149
15 0.0482335854893665
16 0.0491545707033738
17 0.0475621871011123
18 0.0534780426550682
19 0.0715660938138719
20 0.0939330263600716

AutoCovariances of X are:

0 1.14693790165038
1 1.09242958215267
2 0.95665329889968
3 0.782050821478561
4 0.60929078392929
5 0.46737962399361
6 0.36495658921123

Time Series and Forecasting

CrossCorrelation Class • 433
AutoCovariances of Y are:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.298426970727032</td>
</tr>
<tr>
<td>1</td>
<td>0.260942845999682</td>
</tr>
<tr>
<td>2</td>
<td>0.244377603086156</td>
</tr>
<tr>
<td>3</td>
<td>0.23942463361545</td>
</tr>
</tbody>
</table>

AutoCorrelations of X are:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>0.952474916541448</td>
</tr>
<tr>
<td>2</td>
<td>0.834092130980584</td>
</tr>
<tr>
<td>3</td>
<td>0.681859776674248</td>
</tr>
<tr>
<td>4</td>
<td>0.531232576318765</td>
</tr>
<tr>
<td>5</td>
<td>0.407502117801518</td>
</tr>
<tr>
<td>6</td>
<td>0.31820082733867</td>
</tr>
<tr>
<td>7</td>
<td>0.260194532151575</td>
</tr>
<tr>
<td>8</td>
<td>0.22751261943721</td>
</tr>
<tr>
<td>9</td>
<td>0.213069602752258</td>
</tr>
<tr>
<td>10</td>
<td>0.208330776250152</td>
</tr>
</tbody>
</table>

AutoCorrelations of Y are:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>0.970756656983059</td>
</tr>
<tr>
<td>2</td>
<td>0.896039613512222</td>
</tr>
<tr>
<td>3</td>
<td>0.79254837483442</td>
</tr>
<tr>
<td>4</td>
<td>0.67996384208558</td>
</tr>
<tr>
<td>5</td>
<td>0.574477588242088</td>
</tr>
<tr>
<td>6</td>
<td>0.485447988483746</td>
</tr>
<tr>
<td>7</td>
<td>0.416079448222429</td>
</tr>
<tr>
<td>8</td>
<td>0.365607377277169</td>
</tr>
<tr>
<td>9</td>
<td>0.33038225602345</td>
</tr>
<tr>
<td>10</td>
<td>0.306520730819207</td>
</tr>
</tbody>
</table>
CrossCorrelation.StdErr Enumeration

Standard Error computation method.

public enumeration Imsl.Stat.CrossCorrelation.StdErr

Fields

Bartletts

Indicates standard error computation using Bartlett’s formula.

BartlettsNoCC

public Imsl.Stat.CrossCorrelation.StdErr BartlettsNoCC
Indicates standard error computation using Bartlett’s formula with the assumption of no cross-correlation.

MultiCrossCorrelation Class

Computes the multichannel cross-correlation function of two mutually stationary multichannel time series.

public class Imsl.Stat.MultiCrossCorrelation

Constructor

MultiCrossCorrelation

public MultiCrossCorrelation(double[,] x, double[,] y, int maximumLag)
Constructor to compute the multichannel cross-correlation function of two mutually stationary multichannel time series.

x A two-dimensional double array containing the first multichannel stationary time series. Each row of x corresponds to an observation of a multivariate time series and each column of x corresponds to a univariate time series.

y A two-dimensional double array containing the second multichannel stationary time series. Each row of y corresponds to an observation of a multivariate time series and each column of y corresponds to a univariate time series.

maximumLag A int containing the maximum lag of the cross-covariance and cross-correlations to be computed. maximumLag must be greater than or equal to 1 and less than the minimum number of observations of x and y.
Methods

GetCrossCorrelation
public double[,][,] GetCrossCorrelation()
Returns the cross-correlations between the channels of x and y.

The cross-correlation between channel i of the x series and channel j of the y series at lag k, where k = -maximumLag, ..., 0, 1, ..., maximumLag, corresponds to output array, CC[k,i,j] where k= 0, 1, ..., (2*maximumLag), i = 1, ..., x.GetLength(1), and j = 1, ..., y.GetLength(1).

Returns — A double array of size 2 * maximumLag +1 by x.GetLength(1) by y.GetLength(1) containing the cross-correlations between the time series x and y.

Imsl.Stat.NonPosVarianceXYException is thrown if the problem is ill-conditioned. The variance is too small to work with.

GetCrossCovariance
public double[,][,] GetCrossCovariance()
Returns the cross-covariances between the channels of x and y.

The cross-covariances between channel i of the x series and channel j of the y series at lag k where k = -maximumLag, ..., 0, 1, ..., maximumLag, corresponds to output array, CCV[k,i,j] where k= 0, 1, ..., (2*maximumLag), i = 1, ..., x.GetLength(1), and j = 1, ..., y.GetLength(1).

Returns — A double array of size 2 * maximumLag +1 by x.GetLength(1) by y.GetLength(1) containing the cross-covariances between the time series x and y.

Imsl.Stat.NonPosVarianceXYException is thrown if the problem is ill-conditioned. The variance is too small to work with.

GetMeanX
public double[] GetMeanX()
Returns an estimate of the mean of each channel of x.

Returns — A one-dimensional double containing the estimate of the mean of each channel in time series x.

GetMeanY
public double[] GetMeanY()
Returns an estimate of the mean of each channel of y.

Returns — A one-dimensional double containing the estimate of the mean of each channel in the time series y.

GetVarianceX
public double[] GetVarianceX()
Returns the variances of the channels of x.

Returns — A one-dimensional double containing the variances of each channel in the time series x.
Imsl.Stat.NonPosVarianceXYException is thrown if the problem is ill-conditioned. The variance is too small to work with.

GetVarianceY

```java
public double[] GetVarianceY()
```

Returns the variances of the channels of y.

Returns — A one-dimensional double containing the variances of each channel in the time series y.

Imsl.Stat.NonPosVarianceXYException is thrown if the problem is ill-conditioned. The variance is too small to work with.

**Description**

MultiCrossCorrelation estimates the multichannel cross-correlation function of two mutually stationary multichannel time series. Define the multichannel time series X by

\[ X = (X_1, X_2, \ldots, X_p) \]

where

\[ X_j = (X_{1j}, X_{2j}, \ldots, X_{nj})^T, \quad j = 1, 2, \ldots, p \]

with \( n = x.GetLength(0) \) and \( p = x.GetLength(1) \). Similarly, define the multichannel time series Y by

\[ Y = (Y_1, Y_2, \ldots, Y_q) \]

where

\[ Y_j = (Y_{1j}, Y_{2j}, \ldots, Y_{mj})^T, \quad j = 1, 2, \ldots, q \]

with \( m = y.GetLength(0) \) and \( q = y.GetLength(1) \). The columns of X and Y correspond to individual channels of multichannel time series and may be examined from a univariate perspective. The rows of X and Y correspond to observations of p-variate and q-variate time series, respectively, and may be examined from a multivariate perspective. Note that an alternative characterization of a multivariate time series X considers the columns to be observations of the multivariate time series while the rows contain univariate time series. For example, see Priestley (1981, page 692) and Fuller (1976, page 14).

Let \( \hat{\mu}_X = xmean \) be the row vector containing the means of the channels of X. In particular,

\[ \hat{\mu}_X = (\hat{\mu}_{X_1}, \hat{\mu}_{X_2}, \ldots, \hat{\mu}_{X_p}) \]

where for \( j = 1, 2, \ldots, p \)

\[ \hat{\mu}_{X_j} = \begin{cases} 
\mu_{X_j} & \text{for } \mu_{X_j} \text{ known} \\
\frac{1}{n} \sum_{t=1}^{n} X_{tj} & \text{for } \mu_{X_j} \text{ unknown}
\end{cases} \]
Let $\hat{\mu}_Y = y_{\text{mean}}$ be similarly defined. The cross-covariance of lag $k$ between channel $i$ of $X$ and channel $j$ of $Y$ is estimated by

$$\hat{\sigma}_{X_iY_j}(k) = \begin{cases} \frac{1}{N} \sum_t (X_{ti} - \hat{\mu}_{X_i})(Y_{t+k,j} - \hat{\mu}_{Y_j}) & k = 0, 1, \ldots, K \\ \frac{1}{N} \sum_t (X_{ti} - \hat{\mu}_{X_i})(Y_{t+k,j} - \hat{\mu}_{Y_j}) & k = -1, -2, \ldots, -K \end{cases}$$

where $i = 1, \ldots, p$, $j = 1, \ldots, q$, and $K = \text{maximumLag}$. The summation on $t$ extends over all possible cross-products with $N$ equal to the number of cross-products in the sum.

Let $\hat{\sigma}_X(0) = x_{\text{var}}$, where $x_{\text{var}}$ is the variance of $X$, be the row vector consisting of estimated variances of the channels of $X$. In particular,

$$\hat{\sigma}_X(0) = (\hat{\sigma}_{X_1}(0), \hat{\sigma}_{X_2}(0), \ldots, \hat{\sigma}_{X_p}(0))$$

where

$$\hat{\sigma}_{X_i}(0) = \frac{1}{n} \sum_{t=1}^{n} (X_{tj} - \hat{\mu}_{X_i})^2, \quad j = 0, 1, \ldots, p$$

Let $\hat{\sigma}_Y(0) = y_{\text{var}}$, where $y_{\text{var}}$ is the variance of $Y$, be similarly defined. The cross-correlation of lag $k$ between channel $i$ of $X$ and channel $j$ of $Y$ is estimated by

$$\hat{\rho}_{X_iY_j}(k) = \frac{\hat{\sigma}_{X_iY_j}(k)}{[\hat{\sigma}_X(0)\hat{\sigma}_Y(0)]^{1/2}} \quad k = 0, \pm 1, \ldots, \pm K$$

**Example 1: MultiCrossCorrelation**

Consider the Wolfer Sunspot Data ($Y$) (Box and Jenkins 1976, page 530) along with data on northern light activity ($X_1$) and earthquake activity ($X_2$) (Robinson 1967, page 204) to be a three-channel time series. Methods `GetCrossCovariance` and `GetCrossCorrelation` are used to compute the cross-covariances and cross-correlations between $X_1$ and $Y$ and between $X_2$ and $Y$ with lags from $-\text{maximumLag} = -10$ through lag $\text{maximumLag} = 10$.

```csharp
using System;
using Imsl.Stat;
using PrintMatrix = Imsl.Math.PrintMatrix;
using Matrix = Imsl.Math.Matrix;

public class MultiCrossCorrelationEx1
{
    public static void Main(String[] args)
    {
        int i;
        double[,] x = {
            {155.0, 66.0}, {113.0, 62.0},
            {3.0, 66.0}, {10.0, 197.0},
            {0.0, 63.0}, {0.0, 0.0},
            {12.0, 121.0}, {86.0, 0.0},
            {102.0, 113.0}, {20.0, 27.0},
```
double[,] y = {{101.0}, {82.0},
{66.0}, {35.0},
{31.0}, {7.0},
{20.0}, {92.0},
{154.0}, {126.0},
{85.0}, {68.0},
{38.0}, {23.0},
{10.0}, {24.0},
{83.0}, {132.0},
{131.0}, {118.0}};
MultiCrossCorrelation mcc =
    new MultiCrossCorrelation(x, y, 10);
    new PrintMatrix("Mean of X : ").Print(mcc.GetMeanX());
    new PrintMatrix("Variance of X : ").Print(mcc.GetVarianceX());
    new PrintMatrix("Mean of Y : ").Print(mcc.GetMeanY());
    new PrintMatrix("Variance of Y : ").Print(mcc.GetVarianceY());

    double[,] tmpArr = new double[x.GetLength(1), y.GetLength(1)];
    double[,] ccv = mcc.GetCrossCovariance();
    Console.Out.WriteLine
        ("Multichannel cross-covariance between X and Y");
    for (i = 0; i < 21; i++)
    {
        for (int j = 0; j < x.GetLength(1); j++)
for (int k=0;k<y.GetLength(1);k++)
    tmpArr[j,k] = ccv[i,j,k];
Console.Out.WriteLine("Lag K = "+(i - 10));
new PrintMatrix("CrossCovariances :").Print(tmpArr);
}
double[,,] cc = mcc.GetCrossCorrelation();
Console.Out.WriteLine("Multichannel cross-correlation between X and Y");
for (i = 0; i < 21; i++)
{
    for (int j=0;j<x.GetLength(1);j++)
    for (int k=0;k<y.GetLength(1);k++)
        tmpArr[j,k] = cc[i,j,k];
    Console.Out.WriteLine("Lag K = "+(i - 10));
    new PrintMatrix("CrossCorrelations :").Print(tmpArr);
}

Output

Mean of X :
0  63.43
1  97.97

Variance of X :
0  2643.6851
1  1978.4291

Mean of Y :
0  46.94

Variance of Y :
0  1383.7564

Multichannel cross-covariance between X and Y
Lag K = -10
CrossCovariances :
0  -20.5123555555557
1   70.7132444444444

Lag K = -9
CrossCovariances :
0  65.0243098901099
1  38.1363054945055
Lag K = -8
CrossCovariances :
  0
  1  135.57832173913

Lag K = -7
CrossCovariances :
  0
  1  100.362230107527

Lag K = -6
CrossCovariances :
  0
  1  44.9678638297872

Lag K = -5
CrossCovariances :
  0
  1 -11.8094631578948

Lag K = -4
CrossCovariances :
  0
  1  32.6926333333334

Lag K = -3
CrossCovariances :
  0
  1 -40.1185092783505

Lag K = -2
CrossCovariances :
  0
  1 -152.649118367347

Lag K = -1
CrossCovariances :
  0
  1 -212.95022020202

Lag K = 0
CrossCovariances :
  0
  1 -104.7518

Lag K = 1
CrossCovariances :
Lag K = 2
CrossCovariances:
0 628.385118367347
1 84.7751673469388

Lag K = 3
CrossCovariances:
0 438.271931958763
1 75.9630371134021

Lag K = 4
CrossCovariances:
0 238.792741666667
1 200.383466666667

Lag K = 5
CrossCovariances:
0 143.621147368421
1 282.986431578947

Lag K = 6
CrossCovariances:
0 252.973774468085
1 234.393289361702

Lag K = 7
CrossCovariances:
0 479.468286021505
1 223.033735483871

Lag K = 8
CrossCovariances:
0 724.912243478261
1 124.456582608696

Lag K = 9
CrossCovariances:
0 924.971232967034
1 -79.5174307692309

Lag K = 10
CrossCovariances:
0 922.759311111112

Time Series and Forecasting MultiCrossCorrelation Class • 443
Multichannel cross-correlation between X and Y

<table>
<thead>
<tr>
<th>Lag K</th>
<th>CrossCorrelations</th>
</tr>
</thead>
<tbody>
<tr>
<td>-10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 -0.0107245938219684</td>
</tr>
<tr>
<td></td>
<td>1 0.0427376557935899</td>
</tr>
<tr>
<td>-9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 0.0339970370656115</td>
</tr>
<tr>
<td></td>
<td>1 0.023048812287829</td>
</tr>
<tr>
<td>-8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 0.113265706453004</td>
</tr>
<tr>
<td></td>
<td>1 0.0819407975561327</td>
</tr>
<tr>
<td>-7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 0.129032618058936</td>
</tr>
<tr>
<td></td>
<td>1 0.060659035081169</td>
</tr>
<tr>
<td>-6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 0.074309566502109</td>
</tr>
<tr>
<td></td>
<td>1 0.0271776680765982</td>
</tr>
<tr>
<td>-5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 0.026506228548632</td>
</tr>
<tr>
<td></td>
<td>1 -0.00713740085770933</td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 0.0380021196855836</td>
</tr>
<tr>
<td></td>
<td>1 0.0197587868528454</td>
</tr>
<tr>
<td>-3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 0.11390192098873</td>
</tr>
<tr>
<td></td>
<td>1 -0.0242468161934945</td>
</tr>
<tr>
<td>-2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 0.186035762912295</td>
</tr>
<tr>
<td></td>
<td>1 -0.0922580420292281</td>
</tr>
</tbody>
</table>
Lag $K = -1$
CrossCorrelations :
0
1  0.303063597562697
1 -0.128702809263875

Lag $K = 0$
CrossCorrelations :
0
0  0.429575382251174
1 -0.0633098708358119

Lag $K = 1$
CrossCorrelations :
0
0  0.423565683647071
1  0.0333377002981115

Lag $K = 2$
CrossCorrelations :
0
0  0.328542235922487
1  0.051236397797642

Lag $K = 3$
CrossCorrelations :
0
0  0.22914425606054
1  0.0459105243818767

Lag $K = 4$
CrossCorrelations :
0
0  0.124849394067548
1  0.121107717407232

Lag $K = 5$
CrossCorrelations :
0
0  0.075090277447643
1  0.171031279954621

Lag $K = 6$
CrossCorrelations :
0
0  0.132263745693782
1  0.14162566889261

Lag $K = 7$
CrossCorrelations :
0
0  0.250683184784367
1  0.134797082107539

Lag $K = 8$
CrossCorrelations:
0
0 0.37901007257894
1 0.0752190432013873

Lag K = 9
CrossCorrelations:
0
0 0.48360807434863
1 -0.0480587280714567

Lag K = 10
CrossCorrelations:
0
0 0.48245160241607
1 -0.168795069078383

---

**ARMA Class**

Computes least-square estimates of parameters for an ARMA model.

```csharp
public class Imsl.Stat.ARMa
```

**Properties**

BackwardOrigin

```csharp
    public int BackwardOrigin {get; set; }
```

The maximum backward origin. An `int` specifying the maximum backward origin.

`BackwardOrigin` must be greater than or equal to 0 and less than or equal to `z.Length - Math.max(maxar, maxma)`, where

`maxar = Math.max(ARLags[i])`, `maxma = Math.max(MALags[j])`, and forecasts at origins `z.Length - BackwardOrigin` through `z.Length` are generated. Default: `BackwardOrigin = 0`.

Center

```csharp
    public bool Center {get; set; }
```

The center option. A `bool` value specifying whether the time series center occurs about its mean.

If `Center` is set to `false`, the time series is not centered about its mean. If `Center` is set to `true`, the time series is centered about its mean. By Default, `Center = false`.

Confidence

```csharp
    public double Confidence {get; set; }
```
The confidence percent probability limits of the forecasts. A double scalar specifying the confidence percent probability limits of the forecasts.
Typical choices for Confidence are 0.90, 0.95, and 0.99. Confidence must be greater than 0.0 and less than 1.0. Default: Confidence = 0.95.

Constant
public double Constant {get; }
Returns the constant parameter estimate. A double scalar containing the constant parameter estimate.

ConvergenceTolerance
public double ConvergenceTolerance {get; set; }
The tolerance level used to determine convergence of the nonlinear least-squares algorithm. A double scalar containing the tolerance level used to determine convergence of the nonlinear least-squares algorithm.
ConvergenceTolerance represents the minimum relative decrease in sum of squares between two iterations required to determine convergence. Hence, ConvergenceTolerance must be greater than or equal to 0. The default value is \( \max(10^{-20}, \text{eps}^{2/3}) \), where \( \text{eps} = 2.2204460492503131e-16 \).

MaxIterations
public int MaxIterations {get; set; }
The maximum number of iterations. An int scalar specifying the maximum number of iterations allowed in the nonlinear equation solver used in both the method of moments and least-squares algorithms.
Default: MaxIterations = 200.

MeanEstimate
public double MeanEstimate {get; set; }
An update of the mean of the time series z. A double scalar containing the mean of the time series z.
If the time series is not centered about its mean, and least-squares algorithm is used, the mean is not used in parameter estimation.

Method
public Imsl.Stat.ARMA.ParamEstimation Method {get; set; }
The method used to estimate the autoregressive and moving average parameters estimates. An int specifying the method used to estimate the autoregressive and moving average parameters estimates.
If ARMA.ParamEstimation.MethodOfMoments is specified, the autoregressive and moving average parameters are estimated by a method of moments procedure.
If ARMA.ParamEstimation.LeastSquares is specified, the autoregressive and moving average parameters are estimated by a least-squares procedure. By default, Method = ARMA.ParamEstimation.MethodOfMoments.

RelativeError
public double RelativeError {get; set; }
The stopping criterion for use in the nonlinear equation solver. A double scalar containing the stopping criterion for use in the nonlinear equation solver used in both the method of moments and least-squares algorithms.

Default: $\text{RelativeError} = 100 \times 2.204460492503131e-16$.

SSResidual

```csharp
public double SSResidual {get; }
```

Returns the sum of squares of the random shock. A double scalar containing the sum of squares of the random shock, $\text{residual}[0]^2 + \ldots + \text{residual}[\text{na} - 1]^2$, where residual is the array returned from the GetResidual method and $\text{na} = \text{residual}.\text{Length}$.

This property is only applicable using least-squares algorithm.

Variance

```csharp
public double Variance {get; }
```

Returns the variance of the time series $\text{z}$. A double scalar containing the variance of the time series $\text{z}$.

**Constructor**

ARMA

```csharp
public ARMA(int p, int q, double[] z)
```

Constructor for ARMA.

- $\text{p}$ An int scalar containing the number of autoregressive (AR) parameters.
- $\text{q}$ An int scalar containing the number of moving average (MA) parameters.
- $\text{z}$ A double array containing the observations.

System.ArgumentException is thrown if $\text{p}$, $\text{q}$, and $\text{z}.\text{Length}$ are not consistent

**Methods**

Compute

```csharp
public void Compute()
```

Computes least-square estimates of parameters for an ARMA model.

- Imsl.Stat.MatrixSingularException is thrown if the input matrix is singular
- Imsl.Stat.TooManyCallsException is thrown if the number of calls to the function has exceeded
- Imsl.Stat.IncreaseErrRelException is thrown if the bound for the relative error is too small
- Imsl.Stat.NewInitialGuessException is thrown if the iteration has not made good progress
- Imsl.Stat.IllConditionedException is thrown if the problem is ill-conditioned
Imsl.Stat.TooManyIterationsException is thrown if the maximum number of iterations exceeded
Imsl.Stat.TooManyFunctionEvaluationsException is thrown if the maximum number of function evaluations exceeded
Imsl.Stat.TooManyJacobianEvalException is thrown if the maximum number of Jacobian evaluations exceeded

Forecast
public double[,] Forecast(int nPredict)
Computes forecasts and their associated probability limits for an ARMA model.

nPredict An int scalar containing the maximum lead time for forecasts. nPredict must be greater than 0.

Returns — A double matrix of dimensions of nPredict by BackwardOrigin+1 containing the forecasts. Return null if the least-square estimates of parameters is not computed.

GetAR
public double[] GetAR()
Returns the final autoregressive parameter estimates.

Returns — A double array of length p containing the final autoregressive parameter estimates.

GetAutoCovariance
public double[] GetAutoCovariance()
Returns the autocovariances of the time series z.

Returns — A double array containing the autocovariances of lag k, where k = 1, ..., p + q + 1.

GetDeviations
public double[] GetDeviations()
Returns the deviations from each forecast that give the confidence percent probability limits.

Returns — A double array of length nPredict containing the deviations from each forecast that give the confidence percent probability limits.

GetMA
public double[] GetMA()
Returns the final moving average parameter estimates.

Returns — A double array of length q containing the final moving average parameter estimates.

GetParamEstimatesCovariance
public double[,] GetParamEstimatesCovariance()
Returns the covariances of parameter estimates.

The ordering of variables is mean, AR, and MA.
Returns — A double matrix of \( np \) by \( np \) dimensions, where \( np = p + q + 1 \) if \( z \) is centered about MeanEstimate, and \( np = p + q \) if \( z \) is not centered, containing the covariances of parameter estimates.

GetPsiWeights

```csharp
public double[] GetPsiWeights()
```

Returns the psi weights of the infinite order moving average form of the model.

Returns — A double array of length \( nPredict \) containing the psi weights of the infinite order moving average form of the model.

GetResidual

```csharp
public double[] GetResidual()
```

Returns the residuals at the final parameter estimate.

This method is only applicable using least-squares algorithm.

Returns — A double array of length \( z.Length - Math.max(arLags[i]) + length \) containing the residuals (including backcasts) at the final parameter estimate point in the first \( z.Length - Math.max(arLags[i]) + nb \), where \( nb \) is the number of values backcast.

SetARLags

```csharp
public void SetARLags(int[] arLags)
```

The order of the autoregressive parameters.

The elements of \( arLags \) must be greater than or equal to 1. Default: \( arLags = [1, 2, ..., p] \)

\[ arLags \quad \text{An int array of length p containing the order of the autoregressive parameters.} \]

SetBackcasting

```csharp
public void SetBackcasting(int length, double tolerance)
```

Sets backcasting option.

\[ length \quad \text{An int scalar containing the maximum length of backcasting and must be}
\quad \text{greater than or equal to 0. Default: } length = 10. \]

\[ tolerance \quad \text{A double scalar containing the tolerance level used to determine}
\quad \text{convergence of the backcast algorithm. Typically, } tolerance \text{ is set to a fraction of}
\quad \text{an estimate of the standard deviation of the time series. Default: } tolerance = 0.01
\quad * \text{standard deviation of } z. \]

SetInitialEstimates

```csharp
public void SetInitialEstimates(double[] ar, double[] ma)
```

Sets preliminary estimates.

\[ ar \quad \text{and } ma \text{ are computed internally if this method is not used. This method is only}
\quad \text{applicable using least-squares algorithm.} \]

\[ ar \quad \text{A double array of length p containing preliminary estimates of the autoregressive}
\quad \text{parameters.} \]

\[ ma \quad \text{A double array of length q containing preliminary estimates of the moving average}
\quad \text{parameters.} \]
public void SetMALags(int[] maLags)

Sets the order of the moving average parameters.

The `maLags` elements must be greater than or equal to 1. Default: `maLags = [1, 2, ..., q]`

`maLags` An int array of length `q` containing the order of the moving average parameters.

### Description

Class `ARMA` computes estimates of parameters for a nonseasonal ARMA model given a sample of observations, \( \{W_t\} \), for \( t = 1, 2, \ldots, n \), where \( n = z.Length \). There are two methods, method of moments and least squares, from which to choose. The default is method of moments.

Two methods of parameter estimation, method of moments and least squares, are provided. The user can choose a method using the `Method` property. If the user wishes to use the least-squares algorithm, the preliminary estimates are the method of moments estimates by default. Otherwise, the user can input initial estimates by using the `SetInitialEstimates` method. The following table lists the appropriate methods and properties for both the method of moments and least-squares algorithm:

<table>
<thead>
<tr>
<th>Least Squares</th>
<th>Both Method of Moment and Least Squares</th>
</tr>
</thead>
<tbody>
<tr>
<td>Center</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td></td>
</tr>
<tr>
<td>RelativeError</td>
<td></td>
</tr>
<tr>
<td>MaxIterations</td>
<td></td>
</tr>
<tr>
<td>MeanEstimate</td>
<td></td>
</tr>
<tr>
<td>MeanEstimate</td>
<td></td>
</tr>
<tr>
<td>AutoCovariance</td>
<td></td>
</tr>
<tr>
<td>Variance</td>
<td></td>
</tr>
<tr>
<td>Constant</td>
<td></td>
</tr>
<tr>
<td>AR</td>
<td></td>
</tr>
<tr>
<td>MA</td>
<td></td>
</tr>
</tbody>
</table>

### Method of Moments Estimation

Suppose the time series \( \{Z_t\} \) is generated by an ARMA \((p, q)\) model of the form

\[
\phi(B)Z_t = \theta_0 + \theta(B)A_t
\]

for \( t \in \{0, \pm 1, \pm 2, \ldots\} \)
Let $\hat{\mu} = \text{MeanEstimate}$ be the estimate of the mean $\mu$ of the time series $\{Z_t\}$, where $\hat{\mu}$ equals the following:

$$\hat{\mu} = \begin{cases} 
\mu & \text{for } \mu \text{ known} \\
\frac{1}{n} \sum_{t=1}^{n} Z_t & \text{for } \mu \text{ unknown}
\end{cases}$$

The autocovariance function is estimated by

$$\hat{\sigma}(k) = \frac{1}{n} \sum_{t=1}^{n-k} (Z_t - \hat{\mu})(Z_{t+k} - \hat{\mu})$$

for $k = 0, 1, \ldots, K$, where $K = p + q$. Note that $\hat{\sigma}(0)$ is an estimate of the sample variance.

Given the sample autocovariances, the function computes the method of moments estimates of the autoregressive parameters using the extended Yule-Walker equations as follows:

$$\hat{\Sigma} \hat{\phi} = \hat{\sigma}$$

where

$$\hat{\phi} = \left( \hat{\phi}_1, \ldots, \hat{\phi}_p \right)^T$$

$$\hat{\Sigma}_{ij} = \hat{\sigma}(|q + i - j|), \ i, j = 1, \ldots, p$$

$$\hat{\sigma}_i = \hat{\sigma}(q + i), \ i = 1, \ldots, p$$

The overall constant $\theta_0$ is estimated by the following:

$$\hat{\theta}_0 = \begin{cases} 
\hat{\mu} & \text{for } p = 0 \\
\hat{\mu} \left( 1 - \sum_{i=1}^{p} \hat{\phi}_i \right) & \text{for } p > 0
\end{cases}$$

The moving average parameters are estimated based on a system of nonlinear equations given $K = p + q + 1$ autocovariances, $\sigma(k)$ for $k = 1, \ldots, K$, and $p$ autoregressive parameters $\phi_i$ for $i = 1, \ldots, p$.  

452 • ARMA Class

IMSL C# Numerical Library
Let $Z_t' = \phi(B)Z_t$. The autocovariances of the derived moving average process $Z_t' = \theta(B)A_t$ are estimated by the following relation:

$$\hat{\sigma}'(k) = \begin{cases} \hat{\sigma}(k) & \text{for } p = 0 \\ \sum_{i=0}^{p} \sum_{j=0}^{p} \hat{\phi}_i \hat{\phi}_j (\hat{\sigma}(|k + i - j|)) & \text{for } p \geq 1, \hat{\phi}_0 \equiv -1 \end{cases}$$

The iterative procedure for determining the moving average parameters is based on the relation

$$\sigma(k) = \begin{cases} (1 + \theta_1^2 + \ldots + \theta_q^2) \sigma_A^2 & \text{for } k = 0 \\ (-\theta_k + \theta_{k+1} + \ldots + \theta_{q-k}\theta_q) \sigma_A^2 & \text{for } k \geq 1 \end{cases}$$

where $\sigma(k)$ denotes the autocovariance function of the original $Z_t$ process.

Let $\tau = (\tau_0, \tau_1, \ldots, \tau_q)^T$ and $f = (f_0, f_1, \ldots, f_q)^T$, where

$$\tau_j = \begin{cases} \sigma_A & \text{for } j = 0 \\ -\theta_j/\tau_0 & \text{for } j = 1, \ldots, q \end{cases}$$

and

$$f_j = \sum_{i=0}^{q-j} \tau_i \tau_{i+j} - \hat{\sigma}'(j) \text{ for } j = 0, 1, \ldots, q$$

Then, the value of $\tau$ at the $(i+1)$-th iteration is determined by the following:

$$\tau^{i+1} = \tau^i - (T^i)^{-1} f^i$$

The estimation procedure begins with the initial value

$$\tau^0 = (\sqrt{\hat{\sigma}'(0)}, 0, \ldots, 0)^T$$

and terminates at iteration $i$ when either $\|f^i\|$ is less than $\text{RelativeError}$ or $i$ equals $\text{MaxIterations}$. The moving average parameter estimates are obtained from the final estimate of $\tau$ by setting

$$\hat{\theta}_j = -\tau_j/\tau_0 \text{ for } j = 1, \ldots, q$$

The random shock variance is estimated by the following:
\[ \hat{\sigma}_A^2 = \begin{cases} 
\hat{\sigma}(0) - \sum_{i=1}^{p} \hat{\phi}(i) & \text{for } q = 0 \\
\tau_0^2 & \text{for } q \geq 0
\end{cases} \]

See Box and Jenkins (1976, pp. 498-500) for a description of a function that performs similar computations.

**Least-squares Estimation**

Suppose the time series \( \{Z_t\} \) is generated by a nonseasonal ARMA model of the form,

\[
\phi(B)(Z_t - \mu) = \theta(B)A_t \quad \text{for } t \in \{0, \pm 1, \pm 2, \ldots\}
\]

where \( B \) is the backward shift operator, \( \mu \) is the mean of \( Z_t \), and

\[
\phi(B) = 1 - \phi_1 B^{l_\phi(1)} - \phi_2 B^{l_\phi(2)} - \ldots - \phi_p B^{l_\phi(p)} \quad \text{for } p \geq 0
\]

\[
\theta(B) = 1 - \theta_1 B^{l_\theta(1)} - \theta_2 B^{l_\theta(2)} - \ldots - \theta_q B^{l_\theta(q)} \quad \text{for } q \geq 0
\]

with \( p \) autoregressive and \( q \) moving average parameters. Without loss of generality, the following is assumed:

\[ 1 \leq l_\phi(1) \leq l_\phi(2) \leq \ldots \leq l_\phi(p) \]

\[ 1 \leq l_\theta(1) \leq l_\theta(2) \leq \ldots \leq l_\theta(q) \]

so that the nonseasonal ARMA model is of order \( (p', q') \), where \( p' = l_\phi(p) \) and \( q' = l_\theta(q) \). Note that the usual hierarchical model assumes the following:

\[ l_\phi(i) = i, 1 \leq i \leq p \]

\[ l_\theta(j) = j, 1 \leq j \leq q \]

Consider the sum-of-squares function
\[ S_T (\mu, \phi, \theta) = \sum_{-T+1}^{n} [A_t]^2 \]

where

\[ [A_t] = E[A_t | (\mu, \phi, \theta, Z)] \]

and \( T \) is the backward origin. The random shocks \( \{A_t\} \) are assumed to be independent and identically distributed

\[ N (0, \sigma_A^2) \]

random variables. Hence, the log-likelihood function is given by

\[ l (\mu, \phi, \theta, \sigma_A) = f (\mu, \phi, \theta) - n \ln (\sigma_A) - \frac{S_T (\mu, \phi, \theta)}{2\sigma_A^2} \]

where \( f(\mu, \phi, \theta) \) is a function of \( \mu, \phi, \) and \( \theta \).

For \( T = 0 \), the log-likelihood function is conditional on the past values of both \( Z_t \) and \( A_t \) required to initialize the model. The method of selecting these initial values usually introduces transient bias into the model (Box and Jenkins 1976, pp. 210-211). For \( T = \infty \), this dependency vanishes, and estimation problem concerns maximization of the unconditional log-likelihood function. Box and Jenkins (1976, p. 213) argue that

\[ S_\infty (\mu, \phi, \theta) / (2\sigma_A^2) \]

dominates

\[ l (\mu, \phi, \theta, \sigma_A^2) \]

The parameter estimates that minimize the sum-of-squares function are called least-squares estimates. For large \( n \), the unconditional least-squares estimates are approximately equal to the maximum likelihood-estimates.

In practice, a finite value of \( T \) will enable sufficient approximation of the unconditional sum-of-squares function. The values of \( [A_T] \) needed to compute the unconditional sum of squares are computed iteratively with initial values of \( Z_t \) obtained by back forecasting. The residuals (including backcasts), estimate of random shock variance, and covariance matrix of
the final parameter estimates also are computed. ARIMA parameters can be computed by using Difference with ARMA.

**Forecasting**

The Box-Jenkins forecasts and their associated probability limits for a nonseasonal ARMA model are computed given a sample of \( n = z.\text{Length}, \{Z_t\} \) for \( t = 1, 2, \ldots, n \).

Suppose the time series \( Z_t \) is generated by a nonseasonal ARMA model of the form

\[
\phi(B)Z_t = \theta_0 + \theta(B)A_t
\]

for \( t \in \{0, \pm 1, \pm 2, \ldots\} \), where \( B \) is the backward shift operator, \( \theta_0 \) is the constant, and

\[
\phi(B) = 1 - \phi_1 B^{l_\phi(1)} - \phi_2 B^{l_\phi(2)} - \ldots - \phi_p B^{l_\phi(p)}
\]

\[
\theta(B) = 1 - \theta_1 B^{l_\theta(1)} - \theta_2 B^{l_\theta(2)} - \ldots - \theta_q B^{l_\theta(q)}
\]

with \( p \) autoregressive and \( q \) moving average parameters. Without loss of generality, the following is assumed:

\[
1 \leq l_\phi(1) \leq l_\phi(2) \leq \ldots \leq l_\phi(p)
\]

\[
1 \leq l_\theta(1) \leq l_\theta(2) \leq \ldots \leq l_\theta(q)
\]

so that the nonseasonal ARMA model is of order \((p', q')\), where \( p' = l_\theta(p) \) and \( q' = l_\theta(q) \). Note that the usual hierarchical model assumes the following:

\[
l_\phi(i) = i, 1 \leq i \leq p
\]

\[
l_\theta(j) = j, 1 \leq j \leq q
\]

The Box-Jenkins forecast at origin \( t \) for lead time \( l \) of \( Z_{t+1} \) is defined in terms of the difference equation

\[
\hat{Z}_t(l) = \theta_0 + \phi_1 [Z_{t+l-l_\phi(1)}] + \ldots + \phi_p [Z_{t+l-l_\phi(p)}]
\]
\begin{align*}
+ \left[ A_{t+l} - \theta_1 \left[ A_{t+l-\theta(1)} \right] \right] & \quad - \left[ A_{t+l} - \theta_1 \right] \left[ A_{t+l-\theta(1)} \right] - \ldots - \theta_q \left[ A_{t+l-\theta(q)} \right] \\
& \quad - \ldots - \theta_1 \left[ A_{t+l-\theta(1)} \right] - \theta_1 \left[ A_{t+l-\theta(1)} \right] - \ldots - \theta_q \left[ A_{t+l-\theta(q)} \right]
\end{align*}

where the following is true:

\[ [Z_{t+k}] = \begin{cases} 
Z_{t+k} & \text{for } k = 0, -1, -2, \ldots \\
\hat{Z}_t (k) & \text{for } k = 1, 2, \ldots
\end{cases} \]

\[ [A_{t+k}] = \begin{cases} 
Z_{t+k} - \hat{Z}_{t+k-1} (1) & \text{for } k = 0, -1, -2, \ldots \\
0 & \text{for } k = 1, 2, \ldots
\end{cases} \]

The 100(1 - \alpha) percent probability limits for \( Z_{t+l} \) are given by

\[ \hat{Z}_t (l) \pm z_{1/2} \left\{ 1 + \sum_{j=1}^{l-1} \psi_j^2 \right\}^{1/2} \sigma_A \]

where \( z_{1-(\alpha/2)} \) is the 100(1 - \alpha/2) percentile of the standard normal distribution

\[ \sigma_A^2 \]

and

\[ \{ \psi_j^2 \} \]

are the parameters of the random shock form of the difference equation. Note that the forecasts are computed for lead times \( l = 1, 2, \ldots, L \) at origins \( t = (n - b), (n - b + 1), \ldots, n \), where \( L = \text{nPredict} \) and \( b = \text{BackwardOrigin} \).

The Box-Jenkins forecasts minimize the mean-square error

\[ E \left[ Z_{t+l} - \hat{Z}_t (l) \right]^2 \]

Also, the forecasts can be easily updated according to the following equation:

\[ \hat{Z}_{t+1} (l) = \hat{Z}_t (l + 1) + \psi_l A_{t+1} \]
This approach and others are discussed in Chapter 5 of Box and Jenkins (1976).

Example 1: ARMA

Consider the Wolfer Sunspot Data (Anderson 1971, p. 660) consisting of the number of sunspots observed each year from 1749 through 1924. The data set for this example consists of the number of sunspots observed from 1770 through 1869. The method of moments estimates $\hat{\theta}_0, \hat{\phi}_1, \hat{\phi}_2,$ and $\hat{\theta}_1$

for the ARMA$(2, 1)$ model

$$z_t = \theta_0 + \phi_1 z_{t-1} + \phi_2 z_{t-2} - \theta_1 A_{t-1} + A_t$$

where the errors $A_t$ are independently normally distributed with mean zero and variance

$$\sigma_A^2$$

using System;
using Imsl.Stat;
using PrintMatrix = Imsl.Math.PrintMatrix;

public class ARMAEx1
{
    public static void Main(String[] args)
    {
        double[] z = new double[]{ 100.8, 81.6, 66.5, 34.8, 30.6,
                                    7, 19.8, 92.5, 154.4, 125.9,
                                    84.8, 68.1, 38.5, 22.8, 10.2,
                                    24.1, 82.9, 132, 130.9, 118.1,
                                    89.9, 66.6, 60, 46.9, 41,
                                    21.3, 16, 6.4, 4.1, 6.8,
                                    14.5, 34, 45, 43.1, 47.5,
                                    42.2, 28.1, 10.1, 8.1, 2.5,
                                    0, 1.4, 5, 12.2, 13.9,
                                    35.4, 45.8, 41.1, 30.4, 23.9,
                                    15.7, 6.6, 4, 1.8, 8.5,
                                    16.6, 36.3, 49.7, 62.5, 67,
                                    71, 47.8, 27.5, 8.5, 13.2,
                                    56.9, 121.5, 138.3, 103.2, 85.8,
                                    63.2, 36.8, 24.2, 10.7, 15,
                                    40.1, 61.5, 98.5, 124.3, 95.9,
                                    66.5, 64.5, 54.2, 39, 20.6,
                                    6.7, 4.3, 22.8, 54.8, 93.8,
                                    95.7, 77.2, 59.1, 44, 47,
                                    30.5, 16.3, 7.3, 37.3, 73.9};
}
ARMA arma = new ARMA(2, 1, z);
arma.RelativeError = 0.0;
arma.MaxIterations = 0;
arma.Compute();

new PrintMatrix("AR estimates are: ").Print(arma.GetAR());
Console.Out.WriteLine();
new PrintMatrix("MA estimate is: ").Print(arma.GetMA());

Output

AR estimates are:
0
0 1.2442577984372
1 -0.575149766040151

MA estimate is:
0
0 -0.124089747872598

Example 2: ARMA

The data for this example are the same as that for Example 1. Preliminary method of moments estimates are computed by default, and the method of least squares is used to find the final estimates. Note that at the end of the output, a warning message appears. In most cases, this warning message can be ignored. There are three general reasons this warning can occur:

- Convergence is declared using the criterion based on tolerance, but the gradient of the residual sum-of-squares function is nonzero. This occurs in this example. Either the message can be ignored or ConvergenceTolerance can be reduced to allow more iterations and a slightly more accurate solution.

- Convergence is declared based on the fact that a very small step was taken, but the gradient of the residual sum-of-squares function was nonzero. This message can usually be ignored. Sometimes, however, the algorithm is making very slow progress and is not near a minimum.

- Convergence is not declared after 100 iterations.

Trying a smaller value for ConvergenceTolerance can help determine what caused the error message.

using System;
using Imsl.Stat;

Time Series and Forecasting ARMA Class • 459
using PrintMatrix = Imsl.Math.PrintMatrix;

public class ARMAEx2
{
    public static void Main(String[] args)
    {
        double[] arInit = new double[]{1.24426e0, -5.75149e-1};
        double[] maInit = new double[]{-1.24094e-1};
        double[] z = new double[]{100.8, 81.6, 66.5, 34.8, 30.6,
                                    7, 19.8, 92.5, 154.4, 125.9,
                                    84.8, 68.1, 38.5, 22.8, 10.2,
                                    24.1, 82.9, 132, 130.9, 118.1,
                                    89.9, 66.6, 60, 46.9, 41,
                                    21.3, 16, 6.4, 4.1, 6.8,
                                    14.5, 34, 45, 43.1, 47.5,
                                    42.2, 28.1, 10.1, 8.1, 2.5,
                                    0, 1.4, 5, 12.2, 13.9,
                                    35.4, 45.8, 41.1, 30.4, 23.9,
                                    15.7, 6.6, 4, 1.8, 8.5,
                                    16.6, 36.3, 49.7, 62.5, 67,
                                    71, 47.8, 27.5, 8.5, 13.2,
                                    56.9, 121.5, 138.3, 103.2, 85.8,
                                    63.2, 36.8, 24.2, 10.7, 15,
                                    40.1, 61.5, 98.5, 124.3, 95.9,
                                    66.5, 64.5, 54.2, 39, 20.6,
                                    6.7, 4.3, 22.8, 54.8, 93.8,
                                    96.7, 77.2, 59.1, 44, 47,
                                    30.5, 16.3, 7.3, 37.3, 73.9};
        ARMA arma = new ARMA(2, 1, z);
        arma.SetInitialEstimates(arInit, maInit);
        arma.ConvergenceTolerance = 0.125;
        arma.MeanEstimate = 46.976;
        arma.Compute();

        new PrintMatrix("AR estimates are: ").Print(arma.GetAR());
        Console.Out.WriteLine();
        new PrintMatrix("MA estimate is: ").Print(arma.GetMA());
    }
}

Output

AR estimates are:
0
 1 1.39325700313638
1 -0.736660553488482

MA estimate is:
0
 0 -0.137145395974998
Imsl.Stat.ARMA: Relative function convergence - Both the scaled actual and predicted reductions in the function are less than or equal to the relative function convergence tolerance "convergence_tolerance" = 0.0645856533065147. Imsl.Stat.ARMA: Least squares estimation of the parameters has failed to converge. Increase "length" and/or "tolerance" and/or "convergence_tolerance". The estimates of the parameters at the last iteration may be used as new starting values.

Example 3: Forecasting

Consider the Wolfer Sunspot Data (Anderson 1971, p. 660) consisting of the number of sunspots observed each year from 1749 through 1924. The data set for this example consists of the number of sunspots observed from 1770 through 1869. Method forecast in class ARMA computes forecasts and 95-percent probability limits for the forecasts for an ARMA(2, 1) model fit using the method of moments option. With BackwardOrigin = 3, Forecast method provides forecasts given the data through 1866, 1867, 1868, and 1869, respectively. The deviations from the forecast for computing probability limits, and the psi weights can be used to update forecasts when more data is available. For example, the forecast for the 102-nd observation (year 1871) given the data through the 100-th observation (year 1869) is 77.21; and 95-percent probability limits are given by 77.21 ± 56.30. After observation 101 ( Z_{101} for year 1870) is available, the forecast can be updated by using

\[ \hat{Z}_t(l) \pm z_{\alpha/2} \left\{ 1 + \sum_{j=1}^{l-1} \psi_j^2 \right\}^{1/2} \sigma_A \]

with the psi weight (\psi_1 = 1.37) and the one-step-ahead forecast error for observation 101(Z_{101} - 83.72) to give the following:

\[ 77.21 + 1.37 \times (Z_{101} - 83.72) \]

Since this updated forecast is one step ahead, the 95-percent probability limits are now given by the forecast ±33.22.

using System;
using Imsl.Stat;
using PrintMatrix = Imsl.Math.PrintMatrix;
using PrintMatrixFormat = Imsl.Math.PrintMatrixFormat;

public class ARMAEx3
{
    public static void Main(String[] args)
    {
        double[] z = new double[] { 100.8, 81.6, 66.5, 34.8, 30.6, 7, 19.8, 92.5, 154.4, 125.9, 84.8, 68.1, 38.5, 22.8, 10.2, 24.1, 82.9, 132, 130.9, 118.1,

        Time Series and Forecasting
```csharp
89.9, 66.6, 60, 46.9, 41,
21.3, 16, 6.4, 4.1, 6.8,
14.5, 34, 45, 43.1, 47.5,
42.2, 28.1, 10.1, 8.1, 2.5,
0, 1.4, 5, 12.2, 13.9,
35.4, 45.8, 41.1, 30.4, 23.9,
15.7, 6.6, 4, 1.8, 8.5,
16.6, 36.3, 49.7, 62.5, 67,
71, 47.8, 27.5, 8.5, 13.2,
56.9, 121.5, 138.3, 103.2, 85.8,
63.2, 36.8, 24.2, 10.7, 15,
40.1, 61.5, 98.5, 124.3, 95.9,
66.5, 64.5, 54.2, 39, 20.6,
6.7, 4.3, 22.8, 54.8, 93.8,
95.7, 77.2, 59.1, 44, 47,
30.5, 16.3, 7.3, 37.3, 73.9;

PrintMatrixFormat pmf = new PrintMatrixFormat();

ARMA arma = new ARMA(2, 1, z);
arma.RelativeError = 0.0;
arma.MaxIterations = 0;
arma.Compute();

Console.Out.WriteLine("Method of Moments initial estimates:");
new PrintMatrix("AR estimates are: ").Print(arma.GetAR());
Console.Out.WriteLine();
new PrintMatrix("MA estimate is: ").Print(arma.GetMA());
arma.BackwardOrigin = 3;

String[] labels = new String[]{"Forecast From 1866",
   "Forecast From 1867",
   "Forecast From 1868",
   "Forecast From 1869"};

pmf.SetColumnLabels(labels);
new PrintMatrix("forecasts: ").Print(pmf, arma.Forecast(12));

String[] devlabel = new String[]{"Dev. for prob. limits"};
pmf.SetColumnLabels(devlabel);
new PrintMatrix().Print(pmf, arma.GetDeviations());

pmf = new PrintMatrixFormat();
String[] psilabel = new String[]{"Psi"};
pmf.SetColumnLabels(psilabel);
new PrintMatrix().Print(pmf, arma.GetPsiWeights());
```

**Output**

```
Method of Moments initial estimates:
AR estimates are:
0
0 1.24425777984372
```
MA estimate is:

0

-0.124089747872598

<table>
<thead>
<tr>
<th></th>
<th>Forecast From 1866</th>
<th>Forecast From 1867</th>
<th>Forecast From 1868</th>
<th>Forecast From 1869</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18.283158907917</td>
<td>16.6150394496618</td>
<td>55.1894123001951</td>
<td>83.7197534904998</td>
</tr>
<tr>
<td>1</td>
<td>28.9181987956671</td>
<td>32.0187807491226</td>
<td>62.7607512897864</td>
<td>77.2093688403491</td>
</tr>
<tr>
<td>2</td>
<td>41.0100391699882</td>
<td>45.8274629395489</td>
<td>61.8923574323663</td>
<td>63.4609431669911</td>
</tr>
<tr>
<td>3</td>
<td>49.938736920847</td>
<td>54.1495649798482</td>
<td>56.4571977708288</td>
<td>50.0988037706777</td>
</tr>
<tr>
<td>4</td>
<td>54.093739010518</td>
<td>56.5623448569975</td>
<td>50.1939146011939</td>
<td>41.3802616809111</td>
</tr>
<tr>
<td>5</td>
<td>54.12827846595</td>
<td>54.7780099487564</td>
<td>45.5268065977356</td>
<td>38.2173792044093</td>
</tr>
<tr>
<td>6</td>
<td>51.781515136941</td>
<td>51.170127554685</td>
<td>43.3220470047205</td>
<td>39.2964055194313</td>
</tr>
<tr>
<td>7</td>
<td>48.841683089504</td>
<td>47.707251668767</td>
<td>43.2630438046995</td>
<td>42.4581235229259</td>
</tr>
<tr>
<td>8</td>
<td>46.5338141013054</td>
<td>45.4736140841138</td>
<td>44.4576955781352</td>
<td>45.77151401381</td>
</tr>
<tr>
<td>9</td>
<td>45.3523540994474</td>
<td>44.686056496231</td>
<td>45.9780860181243</td>
<td>48.0757645397578</td>
</tr>
<tr>
<td>10</td>
<td>45.210284250337</td>
<td>44.9908279786143</td>
<td>47.182739634897</td>
<td>49.0371504177457</td>
</tr>
<tr>
<td>11</td>
<td>45.7128292416607</td>
<td>45.8229896119653</td>
<td>47.8071878011807</td>
<td>48.9080731249673</td>
</tr>
</tbody>
</table>

Dev. for prob. limits

<table>
<thead>
<tr>
<th></th>
<th>Dev. for prob. limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>33.2179148279538</td>
</tr>
<tr>
<td>1</td>
<td>56.297995631143</td>
</tr>
<tr>
<td>2</td>
<td>67.6167546802611</td>
</tr>
<tr>
<td>3</td>
<td>70.6432170684592</td>
</tr>
<tr>
<td>4</td>
<td>70.7514758474662</td>
</tr>
<tr>
<td>5</td>
<td>71.0868521382172</td>
</tr>
<tr>
<td>6</td>
<td>71.9073814246285</td>
</tr>
<tr>
<td>7</td>
<td>72.5336378185077</td>
</tr>
<tr>
<td>8</td>
<td>72.74980142406</td>
</tr>
<tr>
<td>9</td>
<td>72.7653184488582</td>
</tr>
<tr>
<td>10</td>
<td>72.7779048168612</td>
</tr>
<tr>
<td>11</td>
<td>72.8225053997691</td>
</tr>
</tbody>
</table>

Psi

<table>
<thead>
<tr>
<th></th>
<th>Psi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.36834752771631</td>
</tr>
<tr>
<td>1</td>
<td>1.12742729085079</td>
</tr>
<tr>
<td>2</td>
<td>0.615805417421561</td>
</tr>
<tr>
<td>3</td>
<td>0.117781138936572</td>
</tr>
<tr>
<td>4</td>
<td>-0.207630243315585</td>
</tr>
<tr>
<td>5</td>
<td>-0.326087340079572</td>
</tr>
<tr>
<td>6</td>
<td>-0.286318223936733</td>
</tr>
<tr>
<td>7</td>
<td>-0.168704620288894</td>
</tr>
<tr>
<td>8</td>
<td>-0.0452361767797933</td>
</tr>
<tr>
<td>9</td>
<td>0.040744958004067</td>
</tr>
<tr>
<td>10</td>
<td>0.0767148074728605</td>
</tr>
<tr>
<td>11</td>
<td>0.0720185429660699</td>
</tr>
</tbody>
</table>
ARMA.ParamEstimation Enumeration

Parameter Estimation procedures.

```csharp
public enumeration Imsl.Stat.ARMA.ParamEstimation

Fields

LeastSquares
    public Imsl.Stat.ARMA.ParamEstimation LeastSquares
    Indicates autoregressive and moving average parameters are estimated by a least-squares procedure.

MethodOfMoments
    public Imsl.Stat.ARMA.ParamEstimation MethodOfMoments
    Indicates autoregressive and moving average parameters are estimated by a method of moments procedure.
```

Difference Class

Differences a seasonal or nonseasonal time series.

```csharp
public class Imsl.Stat.Difference

Property

ObservationsLost
    public int ObservationsLost {get; }
    Returns the number of observations lost because of differencing the time series. An int containing the number of observations lost because of differencing the time series z.

Constructor

Difference
    public Difference()
    Constructor for Difference.
```
Methods

Compute
public double[] Compute(double[] z, int[] periods)
Computes a Difference series.

z A double array containing the time series.
periods A int array containing the periods at which z is to be differenced.

Returns — A double array containing the differenced series.

ExcludeFirst
public void ExcludeFirst(bool exclude)
Excludes observations lost due to differencing.

If set to true, the observations lost due to differencing will be excluded. The differenced series will be the length of the number of observations minus the number of observations lost. If set to false, the observations lost due to differencing will be set to NaN (Not a number) and included in the differenced series. The default is to set the lost observations to NaN.

exclude A boolean specifying whether or not to exclude lost observations due to differencing.

SetOrders
public void SetOrders(int[] orders)
Sets the orders for the Difference object.

The elements of orders must be greater than or equal to 0.

orders An int array of length equal to length of periods, containing the order of each difference given in periods.

Description

Class Difference performs \( m = \text{periods.Length} \) successive backward differences of period \( s_i = \text{periods}[i - 1] \) and order \( d_i = \text{orders}[i - 1] \) for \( i = 1, \ldots, m \) on the \( n = z.Length \) observations \( \{Z_t\} \) for \( t = 1, 2, \ldots, n \).

Consider the backward shift operator \( B \) given by

\[
B^k Z_t = Z_{t-k}
\]

for all \( k \). Then, the backward difference operator with period \( s \) is defined by the following:

\[
\Delta_s Z_t = (1 - B^s) Z_t = Z_t - Z_{t-s} \quad \text{for } s \geq 0
\]
Note that $B_s Z_t$ and $\Delta_s Z_t$ are defined only for $t = (s+1), \ldots, n$. Repeated differencing with period $s$ is simply

$$\Delta_s^d Z_t = (1 - B_s)^d Z_t = \sum_{j=0}^{d} \frac{d!}{j!(d-j)!} (-1)^j B^{sj} Z_t$$

where $d \geq 0$ is the order of differencing. Note that

$$\Delta_s^d Z_t$$

is defined only for $t = (sd + 1), \ldots, n$.

The general difference formula used in the class Difference is given by

$$W_T = \begin{cases} 
\text{NaN} & \text{for } t = 1, \ldots, n_L \\
\Delta_s^{d_1} \Delta_s^{d_2} \ldots \Delta_s^{d_m} Z_t & \text{for } t = n_L + 1, \ldots, n 
\end{cases}$$

where $n_L$ represents the number of observations "lost" because of differencing and NaN represents the missing value code. Note that

$$n_L = \sum_j s_j d_j$$

A homogeneous, stationary time series can be arrived at by appropriately differencing a homogeneous, nonstationary time series (Box and Jenkins 1976, p. 85). Preliminary application of an appropriate transformation followed by differencing of a series can enable model identification and parameter estimation in the class of homogeneous stationary autoregressive moving average models.

**Example 1: Difference**

This example uses the Airline Data (Box and Jenkins 1976, p. 531) consisting of the monthly total number of international airline passengers from January 1949 through December 1960. Difference is used to compute ...

$$W_t = \Delta_1 \Delta_{12} Z_t = (Z_t - Z_{t-12}) - (Z_{t-1} - Z_{t-13})$$

for $t = 14, 15, \ldots, 24$.

using System;
using Imsl.Stat;
public class DifferenceEx1
{
    public static void Main(String[] args)
    {
        int[] periods = new int[]{1, 12};
        int nLost;
        double[] z = new double[]{112.0, 118.0, 132.0, 129.0, 121.0,
                                  135.0, 148.0, 148.0, 136.0, 119.0,
                                  104.0, 118.0, 115.0, 126.0, 141.0,
                                  135.0, 125.0, 149.0, 170.0, 170.0,
                                  158.00, 133.0, 114.0, 140.0};

        Difference diff = new Difference();
        double[] out_Renamed = diff.Compute(z, periods);
        nLost = diff.ObservationsLost;

        Console.Out.WriteLine("Observations Lost = " + nLost);
        for (int i = 0; i < out_Renamed.Length; i++)
            Console.Out.WriteLine(out_Renamed[i]);
    }
}

Output

Observations Lost = 13
NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN 5 1 -3 -2 10 8 0 0 -8 -4 12
Example 2: Difference

This example uses the same data as Example 1. The first number of lost observations are excluded from W due to differencing, and the number of lost observations is also output.

```csharp
using System;
using Imsl.Stat;

public class DifferenceEx2
{
    public static void Main(String[] args)
    {
        int[] periods = new int[]{1, 12};
        int nLost;
        double[] z = new double[]{112.0, 118.0, 132.0, 129.0, 121.0,
                                  135.0, 148.0, 148.0, 136.0, 119.0,
                                  104.0, 118.0, 115.0, 126.0, 141.0,
                                  135.0, 125.0, 149.0, 170.0, 170.0,
                                  158.00, 133.0, 114.0, 140.0};

        Difference diff = new Difference();
        diff.ExcludeFirst(true);
        double[] out_Renamed = diff.Compute(z, periods);
        nLost = diff.ObservationsLost;

        Console.Out.WriteLine
        ("The number of observation lost = " + nLost);
        for (int i = 0; i < out_Renamed.Length; i++)
            Console.Out.WriteLine(out_Renamed[i]);
    }
}
```

Output

The number of observation lost = 13
5
1
-3
-2
10
8
0
0
-8
-4
12
GARCH Class

Computes estimates of the parameters of a GARCH(p,q) model.

public class Imsl.Stat.GARCH

Properties

Akaike

public double Akaike {get; }
Returns the value of Akaike Information Criterion evaluated at the estimated parameter array. A double scalar containing the value of Akaike Information Criterion evaluated at the estimated parameter array.

LogLikelihood

public double LogLikelihood {get; }
Returns the value of Log-likelihood function evaluated at the estimated parameter array. A double scalar containing the value of Log-likelihood function evaluated at the estimated parameter array.

MaxSigma

public double MaxSigma {get; set; }
The value of the upperbound on the first element (sigma) of the array of returned estimated coefficients. A double scalar containing the value of the upperbound on the first element (sigma) of the array of returned estimated coefficients.
Default = 10.

Sigma

public double Sigma {get; }
Returns the estimated value of sigma squared. A double scalar containing the estimated value of sigma squared.

Constructor

GARCH

public GARCH(int p, int q, double[] y, double[] xguess)
Constructor for GARCH.

p  A int scalar containing the number of autoregressive (AR) parameters.
q  A int scalar containing the number of moving average (MA) parameters.
y  A double array containing the observed time series data.
xguess  A double array of length p + q + 1 containing the initial values for the parameter array.
System.ArgumentException is thrown if the dimensions of y, and xguess are not consistent

Methods

Compute
public void Compute()
Computes estimates of the parameters of a GARCH(p,q) model.

Imsl.Stat.ConstrInconsistentException is thrown if the equality constraints are inconsistent

Imsl.Stat.EqConstrInconsistentException is thrown if the equality constraints and the bounds on the variables are found to be inconsistent

Imsl.Stat.NoVectorXException is thrown if no vector X satisfies all of the constraints

Imsl.Stat.TooManyFunctionEvaluationsException is thrown if the number of function evaluations exceeded 1000

Imsl.Stat.VarsDeterminedException is thrown if the variables are determined by the equality constraints

GetAR
public double[] GetAR()
Returns the estimated values of autoregressive (AR) parameters.

Returns — A double array of size p containing the estimated values of autoregressive (AR) parameters.

GetMA
public double[] GetMA()
Returns the estimated values of moving average (MA) parameters.

Returns — A double array of size q containing the estimated values of moving average (MA) parameters.

GetVarCovarMatrix
public double[,] GetVarCovarMatrix()
Returns the variance-covariance matrix.

Returns — A double matrix of size p + q + 1 by p + q + 1 containing the variance-covariance matrix.

GetX
public double[] GetX()
Returns the estimated parameter array, x.

Returns — A double array of size p + q + 1 containing the estimated values of sigma squared, the AR parameters, and the MA parameters.
The Generalized Autoregressive Conditional Heteroskedastic (GARCH) model is defined as

$$y_t = z_t \sigma_t$$

$$\sigma_t^2 = \sigma^2 + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2 + \sum_{i=1}^{q} \alpha_i y_{t-i}^2$$

where $z_t$’s are independent and identically distributed standard normal random variables,

$$\sigma > 0, \beta_i \geq 0, \alpha_i \geq 0$$

and

$$\sum_{i=1}^{p} \beta_i + \sum_{i=1}^{q} \alpha_i < 1$$

The above model is denoted as GARCH(p, q). The $p$ is the autoregressive lag and the $q$ is the moving average lag. When $\beta_i = 0, i = 1, 2, \ldots, p$, the above model reduces to ARCH(q) which was proposed by Engle (1982). The nonnegativity conditions on the parameters implied a nonnegative variance and the condition on the sum of the $\beta_i$’s and $\alpha_i$’s is required for wide sense stationarity.

In the empirical analysis of observed data, GARCH(1,1) or GARCH(1,2) models have often found to appropriately account for conditional heteroskedasticity (Palm 1996). This finding is similar to linear time series analysis based on ARMA models.

It is important to notice that for the above models positive and negative past values have a symmetric impact on the conditional variance. In practice, many series may have strong asymmetric influence on the conditional variance. To take into account this phenomena, Nelson (1991) put forward Exponential GARCH (EGARCH). Lai (1998) proposed and studied some properties of a general class of models that extended linear relationship of the conditional variance in ARCH and GARCH into nonlinear fashion.

The maximum likelihood method is used in estimating the parameters in GARCH(p, q). The log-likelihood of the model for the observed series $\{Y_t\}$ with length $m$ is

$$\log(L) = \frac{m}{2} \log(2\pi) - \frac{1}{2} \sum_{t=1}^{m} y_t^2 / \sigma_t^2 - \frac{1}{2} \sum_{t=1}^{m} \log \sigma_t^2.$$
where \( \sigma_t^2 = \sigma^2 + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2 + \sum_{i=1}^{q} \alpha_i y_{t-i}^2. \)

In the model, if \( q = 0 \), the model GARCH is singular such that the estimated Hessian matrix \( H \) is singular.

The initial values of the parameter array \( x[\cdot] \) entered in array \( \mathbf{xguess} \) must satisfy certain constraints. The first element of \( \mathbf{xguess} \) refers to sigma and must be greater than zero and less than \( \text{MaxSigma} \). The remaining \( p+q \) initial values must each be greater than or equal to zero but less than one.

To guarantee stationarity in model fitting,

\[
\sum_{i=1}^{p+q} x(i) < 1,
\]

is checked internally. The initial values should be selected from the values between zero and one. The value of Akaike Information Criterion is computed by

\[
2 \times \log(L) + 2 \times (p + q + 1),
\]

where \( \log(L) \) is the value of the log-likelihood function at the estimated parameters.

In fitting the optimal model, the class Imsl.Math.MinConGenLin (p. 144), is modified to find the maximal likelihood estimates of the parameters in the model. Statistical inferences can be performed outside of the class GARCH based on the output of the log-likelihood function (LogLikelihood property), the Akaike Information Criterion (Akaike property), and the variance-covariance matrix (GetVarCovarMatrix method).

**Example: GARCH**

The data for this example are generated to follow a GARCH\((p,q)\) process by using a random number generation function \( \text{sgarch} \). The data set is analyzed and estimates of sigma, the AR parameters, and the MA parameters are returned. The values of the Log-likelihood function and the Akaike Information Criterion are returned.

```csharp
using System;
using Imsl.Stat;
using PrintMatrix = Imsl.Math.PrintMatrix;
public class GARCHEx1
{
    static private void sgarch(int p, int q, int m, double[] x, double[] y, double[] z, double[] y0, double[] sigma)
```
int i, j, l;
double s1, s2, s3;
rand.Multiplier = 16807;

for (i = 0; i < m + 1000; i++)
   z[i] = rand.NextNormal();

l = System.Math.Max(p, q);
l = System.Math.Max(l, 1);
for (i = 0; i < l; i++)
   y0[i] = z[i] * x[0];

/* COMPUTE THE INITIAL VALUE OF SIGMA */
s3 = 0.0;
if (System.Math.Max(p, q) >= 1)
{
   for (i = 1; i < (p + q + 1); i++)
      s3 += x[i];
}
for (i = 0; i < l; i++)
   sigma[i] = x[0] / (1.0 - s3);
for (i = l; i < (m + 1000); i++)
{
   s1 = 0.0;
   s2 = 0.0;
   if (q >= 1)
   {
      for (j = 0; j < q; j++)
         s1 += x[j + 1] * y0[i - j - 1] * y0[i - j - 1];
   }
   if (p >= 1)
   {
      for (j = 0; j < p; j++)
         s2 += x[q + 1 + j] * sigma[i - j - 1];
   }
   sigma[i] = x[0] + s1 + s2;
   y0[i] = z[i] * Math.Sqrt(sigma[i]);
}

/*
 * DISCARD THE FIRST 1000 SIMULATED OBSERVATIONS
 */
for (i = 0; i < m; i++)
   y[i] = y0[1000 + i];
return ;

public static void Main(String[] args)
{
   int n, p, q, m;
   double[] x = new double[]{1.3, 0.2, 0.3, 0.4};
   double[] xguess = new double[]{1.0, 0.1, 0.2, 0.3};
   double[] y = new double[1000];
   double[] wk1 = new double[2000];
Output

Sigma estimate is 1.692

AR estimate is
0
0 0.244996117092089
1 0.337235176981931

MR estimate is
0
0 0.309586601528879

Log-likelihood function value is -2707.073
Akaike Information Criterion value is 5422.146

KalmanFilter Class

Performs Kalman filtering and evaluates the likelihood function for the state-space model.

public class Imsl.Stat.KalmanFilter
Properties

LogDeterminant

```csharp
public double LogDeterminant {get; }
```

Returns the natural log of the product of the nonzero eigenvalues of \( P \) where \( P \cdot \sigma^2 \) is the variance-covariance matrix of the observations. A `double` scalar containing the natural log of the product of the nonzero eigenvalues of \( P \) where \( P \cdot \sigma^2 \) is the variance-covariance matrix of the observations.

In the usual case when \( P \) is nonsingular, `LogDeterminant` is the natural log of the determinant of \( P \).

Rank

```csharp
public int Rank {get; }
```

Returns the rank of the variance-covariance matrix for all the observations. A `int` scalar containing the rank of the variance-covariance matrix for all the observations.

SumOfSquares

```csharp
public double SumOfSquares {get; }
```

Returns the generalized sum of squares. A `double` scalar containing the generalized sum of squares.

The estimate of \( \sigma^2 \) is given by `sumOfSquares / n`.

Tolerance

```csharp
public double Tolerance {get; set; }
```

The tolerance used in determining linear dependence. A `double` scalar containing the tolerance used in determining linear dependence.

Default: `tolerance = 100.0*2.2204460492503131e-16`.

Constructor

```csharp
public KalmanFilter(double[] b, double[] covb, int rank, double sumOfSquares, double logDeterminant)
```

Constructor for `KalmanFilter`.

- `b` is the estimated state vector at time \( k \) given the observations through time \( k-1 \).

- `b` is a `double` array containing the estimated state vector.

- `covb` is a `double` array of size `b.Length` by `b.Length` such that `covb * \sigma^2` is the mean squared error matrix for `b`.

- `rank` is an `int` scalar containing the rank of the variance-covariance matrix for all the observations.

- `sumOfSquares` is a `double` scalar containing the generalized sum of squares.

- `logDeterminant` is a `double` scalar containing the natural log of the product of the nonzero eigenvalues of \( P \) where \( P \cdot \sigma^2 \) is the variance-covariance matrix of the observations.
System.ArgumentException is thrown if the dimensions of \( \mathbf{b} \), and \( \mathbf{covb} \) are not consistent.

**Methods**

**Filter**
```
public void Filter()
```
Performs Kalman filtering and evaluates the likelihood function for the state-space model.

**GetCovB**
```
public double[] GetCovB()
```
Returns the mean squared error matrix for \( \mathbf{b} \) divided by sigma squared.

Returns — A double array of size \( \mathbf{b}.\text{Length} \) by \( \mathbf{b}.\text{Length} \) such that \( \mathbf{covb} \ast \sigma^2 \) is the mean squared error matrix for \( \mathbf{b} \).

**GetCovV**
```
public double[,] GetCovV()
```
Returns the variance-covariance matrix of \( \mathbf{v} \) divided by sigma squared.

Returns — A double matrix containing a \( \mathbf{y}.\text{Length} \) by \( \mathbf{y}.\text{Length} \) matrix such that \( \mathbf{covv} \ast \sigma^2 \) is the variance-covariance matrix of \( \mathbf{v} \).

**GetPredictionError**
```
public double[] GetPredictionError()
```
Returns the one-step-ahead prediction error.

Returns — A double array of size \( \mathbf{y}.\text{Length} \) containing the one-step-ahead prediction error.

**GetStateVector**
```
public double[] GetStateVector()
```
Returns the estimated state vector at time \( k + 1 \) given the observations through time \( k \).

Returns — A double array containing the estimated state vector at time \( k + 1 \) given the observations through time \( k \).

**SetQ**
```
public void SetQ(double[,] q)
```
Sets the Q matrix.

Default: There is no error term in the state equation.

\( q \) — A double matrix containing the \( \mathbf{b}.\text{Length} \) by \( \mathbf{b}.\text{Length} \) matrix such that \( q \ast \sigma^2 \) is the variance-covariance matrix of the error vector in the state equation.

**SetTransitionMatrix**
```
public void SetTransitionMatrix(double[,] t)
```
Sets the transition matrix.

Default: \( t = \text{identity matrix} \)
t  A double matrix containing the b.Length by b.Length transition matrix in the state equation.

Update

public void Update(double[] y, double[,] z, double[,] r)
Performs computation of the update equations.

σ² is a positive unknown scalar. Only elements in the upper triangle of r are referenced.

y  A double array containing the observations.

z  A double matrix containing the y.Length by b.Length matrix relating the observations to the state vector in the observation equation.

r  A double matrix containing the y.Length by y.Length matrix such that r * σ² is the variance-covariance matrix of errors in the observation equation.

Description

Class KalmanFilter is based on a recursive algorithm given by Kalman (1960), which has come to be known as the KalmanFilter. The underlying model is known as the state-space model. The model is specified stage by stage where the stages generally correspond to time points at which the observations become available. KalmanFilter avoids many of the computations and storage requirements that would be necessary if one were to process all the data at the end of each stage in order to estimate the state vector. This is accomplished by using previous computations and retaining in storage only those items essential for processing of future observations.

The notation used here follows that of Sallas and Harville (1981). Let yₖ (input in y using method Update) be the nₖ × 1 vector of observations that become available at time k. The subscript k is used here rather than t, which is more customary in time series, to emphasize that the model is expressed in stages k = 1, 2, ... and that these stages need not correspond to equally spaced time points. In fact, they need not correspond to time points of any kind. The observation equation for the state-space model is

\[ y_k = Z_k b_k + e_k \quad k = 1, 2, \ldots \]

Here, Zₖ (input in z using method update) is an nₖ × q known matrix and bₖ is the q × 1 state vector. The state vector bₖ is allowed to change with time in accordance with the state equation

\[ b_{k+1} = T_{k+1} b_k + w_{k+1} \quad k = 1, 2, \ldots \]

starting with b₁ = μ₁ + w₁.

The change in the state vector from time k to k + 1 is explained in part by the transition matrix Tₖ₊₁ (the identity matrix by default, or optionally using method SetTransitionMatrix).
which is assumed known. It is assumed that the \( q \)-dimensional \( w \)\(_k\)s \((k = 1, 2, \ldots)\) are independently distributed multivariate normal with mean vector \( 0 \) and variance-covariance matrix \( \sigma^2 Q_k \), that the \( n_k \)-dimensional \( e \)\(_k\)s \((k = 1, 2, \ldots)\) are independently distributed multivariate normal with mean vector \( 0 \) and variance-covariance matrix \( \sigma^2 R_k \), and that the \( w \)\(_k\)s and \( e \)\(_k\)s are independent of each other. Here, \( \mu_1 \) is the mean of \( b_1 \) and is assumed known, \( \sigma^2 \) is an unknown positive scalar. \( Q_{k+1} \) (input in \( Q \)) and \( R_k \) (input in \( R \)) are assumed known.

Denote the estimator of the realization of the state vector \( b_k \) given the observations \( y_1, y_2, \ldots, y_j \) by

\[
\hat{\beta}_{k|j}
\]

By definition, the mean squared error matrix for

\[
\hat{\beta}_{k|j}
\]

is

\[
\sigma^2 C_{k|j} = E(\hat{\beta}_{k|j} - b_k)(\hat{\beta}_{k|j} - b_k)^T
\]

At the time of the \( k \)-th invocation, we have

\[
\hat{\beta}_{k|k-1}
\]

and

\( C_{k|k-1} \), which were computed from the \( k \)-1-st invocation, input in \( b \) and \( \text{cov} b \), respectively. During the \( k \)-th invocation, \texttt{KalmanFilter} computes the filtered estimate

\[
\hat{\beta}_{k|k}
\]

along with \( C_{k|k} \). These quantities are given by the \textit{update equations}:

\[
\hat{\beta}_{k|k} = \hat{\beta}_{k|k-1} + C_{k|k-1} Z_k^T H_k^{-1} v_k
\]

\[
C_{k|k} = C_{k|k-1} - C_{k|k-1} Z_k^T H_k^{-1} Z_k C_{k|k-1}
\]
where

\[ v_k = y_k - Z_k \hat{\beta}_{k|k-1} \]

and where

\[ H_k = R_k + Z_k C_{k|k-1} Z_k^T \]

Here, \( v_k \) (stored in \( v \)) is the one-step-ahead prediction error, and \( \sigma^2 H_k \) is the variance-covariance matrix for \( v_k \). \( H_k \) is stored in \( \text{covv} \). The "start-up values" needed on the first invocation of \( \text{KalmanFilter} \) are

\[ \hat{\beta}_{1|0} = \mu_1 \]

and \( C_{1|0} = Q_1 \) input via \( b \) and \( \text{covb} \), respectively. Computations for the \( k \)-th invocation are completed by \( \text{KalmanFilter} \) computing the one-step-ahead estimate

\[ \hat{\beta}_{k+1|k} \]

along with \( C_{k+1|k} \) given by the prediction equations:

\[ \hat{\beta}_{k+1|k} = T_{k+1} \hat{\beta}_{k|k} \]

\[ C_{k+1|k} = T_{k+1} C_{k|k} T_{k+1}^T + Q_{k+1} \]

If both the filtered estimates and one-step-ahead estimates are needed by the user at each time point, \( \text{KalmanFilter} \) can be used twice for each time point-first without methods \( \text{SetTransitionMatrix} \) and \( \text{SetQ} \) to produce

\[ \hat{\beta}_{k|k} \]

and \( C_{k|k} \), and second without method \( \text{Update} \) to produce

\[ \hat{\beta}_{k+1|k} \]
and $C_{k+1|k}$ (Without methods SetTransitionMatrix and SetQ, the prediction equations are skipped. Without method update, the update equations are skipped.).

Often, one desires the estimate of the state vector more than one-step-ahead, i.e., an estimate of

$$\hat{\beta}_{k|j}$$

is needed where $k > j + 1$. At time $j$, KalmanFilter is invoked with method Update to compute

$$\hat{\beta}_{j+1|j}$$

Subsequent invocations of KalmanFilter without method Update can compute

$$\hat{\beta}_{j+2|j}, \hat{\beta}_{j+3|j}, \ldots, \hat{\beta}_{k|j}$$

Computations for

$$\hat{\beta}_{k|j}$$

and $C_{k|j}$ assume the variance-covariance matrices of the errors in the observation equation and state equation are known up to an unknown positive scalar multiplier, $\sigma^2$. The maximum likelihood estimate of $\sigma^2$ based on the observations $y_1, y_2, \ldots, y_m$, is given by

$$\hat{\sigma}^2 = SS/N$$

where

$$N = \sum_{k=1}^{m} n_k \quad \text{and} \quad SS = \sum_{k=1}^{m} v_k^T H_k^{-1} v_k$$

$N$ and $SS$ are the input/output arguments n and sumOfSquares.

If $\sigma^2$ is known, the $R_k$s and $Q_k$s can be input as the variance-covariance matrices exactly. The earlier discussion is then simplified by letting $\sigma^2 = 1$.

In practice, the matrices $T_k$, $Q_k$, and $R_k$ are generally not completely known. They may be known functions of an unknown parameter vector $\theta$. In this case, KalmanFilter can be used in conjunction with an optimization class (see class MinUncomMultiVar, IMSL C# Library Math namespace), to obtain a maximum likelihood estimate of $\theta$. The natural logarithm of the likelihood function for $y_1, y_2, \ldots, y_m$ differs by no more than an additive constant from
\[ L(\theta, \sigma^2; y_1, y_2, \ldots, y_m) = -\frac{1}{2} N \ln \sigma^2 - \frac{1}{2} \sum_{k=1}^{m} \ln |\text{det}(H_k)| - \frac{1}{2} \sigma^{-2} \sum_{k=1}^{m} v_k^T H_k^{-1} v_k \]

(Harvey 1981, page 14, equation 2.21).

Here,

\[ \sum_{k=1}^{m} \ln |\text{det}(H_k)| \]

(stored in logDeterminant) is the natural logarithm of the determinant of \( V \) where \( \sigma^2 V \) is the variance-covariance matrix of the observations.

Minimization of \(-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)\) over all \( \theta \) and \( \sigma^2 \) produces maximum likelihood estimates. Equivalently, minimization of \(-2L_c(\theta; y_1, y_2, \ldots, y_m)\) where

\[ L_c(\theta; y_1, y_2, \ldots, y_m) = -\frac{1}{2} N \ln \left( \frac{SS}{N} \right) - \frac{1}{2} \sum_{k=1}^{m} \ln |\text{det}(H_k)| \]

produces maximum likelihood estimates

\[ \hat{\theta} \text{ and } \hat{\sigma}^2 = SS/N \]

Minimization of \(-2L_c(\theta; y_1, y_2, \ldots, y_m)\) instead of \(-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)\), reduces the dimension of the minimization problem by one. The two optimization problems are equivalent since

\[ \hat{\sigma}^2(\theta) = SS(\theta)/N \]

minimizes \(-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)\) for all \( \theta \), consequently,

\[ \hat{\sigma}^2(\theta) \]

can be substituted for \( \sigma^2 \) in \( L(\theta, \sigma^2; y_1, y_2, \ldots, y_m) \) to give a function that differs by no more than an additive constant from \( L_c(\theta; y_1, y_2, \ldots, y_m) \).

The earlier discussion assumed \( H_k \) to be nonsingular. If \( H_k \) is singular, a modification for singular distributions described by Rao (1973, pages 527-528) is used. The necessary changes in the preceding discussion are as follows:

- Replace \( H_k^{-1} \) by a generalized inverse.
• Replace \( \det(H_k) \) by the product of the nonzero eigenvalues of \( H_k \).
• Replace \( N \) by \( \sum_{k=1}^{m} \text{rank}(H_k) \)


**Example: Kalman Filter**

KalmanFilter is used to compute the filtered estimates and one-step-ahead estimates for a scalar problem discussed by Harvey (1981, pages 116-117). The observation equation and state equation are given by

\[
y_k = b_k + \epsilon_k
\]

\[
b_{k+1} = b_k + w_{k+1}
\]

\( k = 1, 2, 3, 4 \)

where the \( \epsilon_k \)s are identically and independently distributed normal with mean 0 and variance \( \sigma^2 \), the \( w_k \)s are identically and independently distributed normal with mean 0 and variance \( 4\sigma^2 \), and \( b_1 \) is distributed normal with mean 4 and variance \( 16\sigma^2 \). Two KalmanFilter objects are needed for each time point in order to compute the filtered estimate and the one-step-ahead estimate. The first object does not use the methods SetTransitionMatrix and SetQ so that the prediction equations are skipped in the computations. The update equations are skipped in the computations in the second object.

This example also computes the one-step-ahead prediction errors. Harvey (1981, page 117) contains a misprint for the value \( v_4 \) that he gives as 1.197. The correct value of \( v_4 = 1.003 \) is computed by KalmanFilter.
double[,] q = {{4}};
double[,] r = {{1}};
double[,] t = {{1}};
double[,] z = {{1}};
double[] ydata = new double[]{4.4, 4.0, 3.5, 4.6};

System.Object[] argFormat =
    new System.Object[] {"k", "j", "b", "cov(b)", "rank", "ss", "ln(det)", "v", "cov(v)"};
Console.Out.WriteLine(String.Format(format, argFormat));

for (int i = 0; i < nobs; i++)
{
    double[] y = new double[]{ydata[i]};
    KalmanFilter kalman =
        new KalmanFilter(b, covb, rank, ss, logDeterminant);
    kalman.Update(y, z, r);
    kalman.Filter();
    b = kalman.GetStateVector();
    covb = kalman.GetCovB();
    rank = kalman.Rank;
    ss = kalman.SumOfSquares;
    logDeterminant = kalman.LogDeterminant;
    double[] v = kalman.GetPredictionError();
    double[,] covv = kalman.GetCovV();
    argFormat[0] = i;
    argFormat[1] = i;
    argFormat[2] = b[0];
    argFormat[3] = covb[0];
    argFormat[5] = ss;
    argFormat[7] = v[0];
    argFormat[8] = covv[0, 0];
    Console.Out.WriteLine(String.Format(format, argFormat));

    kalman =
        new KalmanFilter(b, covb, rank, ss, logDeterminant);
    kalman.SetTransitionMatrix(t);
    kalman.SetQ(q);
    kalman.Filter();
    b = kalman.GetStateVector();
    covb = kalman.GetCovB();
    rank = kalman.Rank;
    ss = kalman.SumOfSquares;
    logDeterminant = kalman.LogDeterminant;
    argFormat[0] = i + 1;
    argFormat[1] = i;
    argFormat[2] = b[0];
    argFormat[3] = covb[0];
    argFormat[5] = ss;
    argFormat[7] = v[0];
    argFormat[8] = covv[0, 0];
    Console.Out.WriteLine(String.Format(format, argFormat));
}

Time Series and Forecasting

KalmanFilter Class • 483
### Output

<table>
<thead>
<tr>
<th>k/j</th>
<th>b</th>
<th>cov(b)</th>
<th>rank</th>
<th>ss</th>
<th>ln(det)</th>
<th>v</th>
<th>cov(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/0</td>
<td>4.376</td>
<td>0.941</td>
<td>1</td>
<td>0.009</td>
<td>2.833</td>
<td>0.400</td>
<td>17.000</td>
</tr>
<tr>
<td>1/0</td>
<td>4.376</td>
<td>4.941</td>
<td>1</td>
<td>0.009</td>
<td>2.833</td>
<td>0.400</td>
<td>17.000</td>
</tr>
<tr>
<td>1/1</td>
<td>4.063</td>
<td>0.832</td>
<td>2</td>
<td>0.033</td>
<td>4.615</td>
<td>-0.376</td>
<td>5.941</td>
</tr>
<tr>
<td>2/1</td>
<td>4.063</td>
<td>4.832</td>
<td>2</td>
<td>0.033</td>
<td>4.615</td>
<td>-0.376</td>
<td>5.941</td>
</tr>
<tr>
<td>2/2</td>
<td>3.597</td>
<td>0.829</td>
<td>3</td>
<td>0.088</td>
<td>6.378</td>
<td>-0.563</td>
<td>5.832</td>
</tr>
<tr>
<td>3/2</td>
<td>3.597</td>
<td>4.829</td>
<td>3</td>
<td>0.088</td>
<td>6.378</td>
<td>-0.563</td>
<td>5.832</td>
</tr>
<tr>
<td>3/3</td>
<td>4.428</td>
<td>0.828</td>
<td>4</td>
<td>0.260</td>
<td>8.141</td>
<td>1.003</td>
<td>5.829</td>
</tr>
<tr>
<td>4/3</td>
<td>4.428</td>
<td>4.828</td>
<td>4</td>
<td>0.260</td>
<td>8.141</td>
<td>1.003</td>
<td>5.829</td>
</tr>
</tbody>
</table>
Chapter 19: Multivariate Analysis

Types

class ClusterKMeans ................................................................. 487
class Dissimilarities ................................................................. 496
class ClusterHierarchical .......................................................... 499
class FactorAnalysis ................................................................. 507
enumeration FactorAnalysis.MatrixType ..................................... 523
enumeration FactorAnalysis.Model ........................................... 524
class DiscriminantAnalysis ........................................................ 525
enumeration DiscriminantAnalysis.Discrimination ....................... 545
enumeration DiscriminantAnalysis.CovarianceMatrix ................. 546
enumeration DiscriminantAnalysis.Classification ....................... 546
enumeration DiscriminantAnalysis.PriorProbabilities .................. 547

Usage Notes

Cluster Analysis

ClusterKMeans performs a K-means cluster analysis. Basic K-means clustering attempts to find a clustering that minimizes the within-cluster sums-of-squares. In this method of clustering the data, matrix X is grouped so that each observation (row in X) is assigned to one of a fixed number, K, of clusters. The sum of the squared difference of each observation about its assigned cluster’s mean is used as the criterion for assignment. In the basic algorithm, observations are transferred from one cluster or another when doing so decreases the within-cluster sums-of-squared differences. When no transfer occurs in a pass through the entire data set, the algorithm stops. ClusterKMeans is one implementation of the basic algorithm.

The usual course of events in K-means cluster analysis is to use ClusterKMeans to obtain the optimal clustering. The clustering is then evaluated by functions described in ”Basic Statistics”, and/or other chapters in this manual. Often, K-means clustering with more than one value of K is performed, and the value of K that best fits the data is used.

Clustering can be performed either on observations or variables. The discussion of the function
ClusterKMeans assumes the clustering is to be performed on the observations, which correspond to the rows of the input data matrix. If variables, rather than observations, are to be clustered, the data matrix should first be transposed. In the documentation for ClusterKMeans, the words "observation" and "variable" are interchangeable.

**Principal Components**

The idea in principal components is to find a small number of linear combinations of the original variables that maximize the variance accounted for in the original data. This amounts to an eigensystem analysis of the covariance (or correlation) matrix. In addition to the eigensystem analysis, when the principal component model is used, FactorAnalysis computes standard errors for the eigenvalues. Correlations of the original variables with the principal component scores also are computed.

**Factor Analysis**

Factor analysis and principal component analysis, while quite different in assumptions, often serve the same ends. Unlike principal components in which linear combinations yielding the highest possible variances are obtained, factor analysis generally obtains linear combinations of the observed variables according to a model relating the observed variable to hypothesized underlying factors, plus a random error term called the unique error or uniqueness. In factor analysis, the unique errors associated with each variable are usually assumed to be independent of the factors. Additionally, in the common factor model, the unique errors are assumed to be mutually independent. The factor analysis model is expressed in the following equation:

\[ x - \mu = \Lambda f + e \]

where \( x \) is the \( p \) vector of observed values, \( \mu \) is the \( p \) vector of variable means, \( \Lambda \) is the \( p \times k \) matrix of factor loadings, \( f \) is the \( k \) vector of hypothesized underlying random factors, \( e \) is the \( p \) vector of hypothesized unique random errors, \( p \) is the number of variables in the observed variables, and \( k \) is the number of factors.

Because much of the computation in factor analysis was originally done by hand or was expensive on early computers, quick (but dirty) algorithms that made the calculations possible were developed. One result is the many factor extraction methods available today. Generally speaking, in the exploratory or model building phase of a factor analysis, a method of factor extraction that is not computationally intensive (such as principal components, principal factor, or image analysis) is used. If desired, a computationally intensive method is then used to obtain the final factors.

**Discriminant Analysis**

The class DiscriminantAnalysis allows linear or quadratic discrimination and the use of either reclassification, split sample, or the leaving-out-one methods in order to evaluate the rule. Moreover, DiscriminantAnalysis can be executed in an online mode, that is, one or more observations can be added to the rule during each invocation of DiscriminantAnalysis.

The mean vectors for each group of observations and an estimate of the common covariance matrix for all groups are input to DiscriminantAnalysis. Output from DiscriminantAnalysis are linear combinations of the observations, which at most separate the groups. These linear combinations may subsequently be used for discriminating between the
groups. Their use in graphically displaying differences between the groups is possibly more important, however.

### ClusterKMeans Class

Perform a $K$-means (centroid) cluster analysis.

```java
public class Imsl.Stat.ClusterKMeans
```

#### Property

**MaxIterations**

```java
public int MaxIterations {get; set; }
```

The maximum number of iterations. A `int` scalar specifying the maximum number of iterations.

Default: `MaxIterations = 30`.

#### Constructor

```java
ClusterKMeans(double[,] x, double[,] cs)
```

Constructor for `ClusterKMeans`.

- **x** A `double` matrix containing the observations to be clustered.
- **cs** A `double` matrix containing the cluster seeds, i.e. estimates for the cluster centers.

`System.ArgumentException` is thrown if `x.GetLength(0)` or `cs.GetLength(0)` are equal 0, or `cs.GetLength(0)` is less than 1

#### Methods

**Compute**

```java
public double[,] Compute()
```

Computes the cluster means.

Returns — A `double` matrix containing computed result.

`Imsl.Stat.NoConvergenceException` is thrown if convergence did not occur within the maximum number of iterations.

`Imsl.Stat.ClusterNoPointsException` is thrown if the cluster seed yields a cluster with no points
GetClusterCounts
  public int[] GetClusterCounts()
  Returns the number of observations in each cluster.
  Returns — An int array containing the number of observations in each cluster.

GetClusterMembership
  public int[] GetClusterMembership()
  Returns the cluster membership for each observation.
  Cluster membership 1 indicates the observation belongs to cluster 1, cluster membership 2 indicates the observation belongs to cluster 2, etc.
  Returns — An int array containing the cluster membership for each observation.

GetClusterSSQ
  public double[] GetClusterSSQ()
  Returns the within sum of squares for each cluster.
  Returns — A double array containing the within sum of squares for each cluster.

SetFrequencies
  public void SetFrequencies(double[] frequencies)
  The frequency for each observation.
  Default: Frequencies[] = 1.
  frequencies A double array of size x.GetLength(0) containing the frequency for each observation.

SetWeights
  public void SetWeights(double[] weights)
  Sets the weight for each observation.
  Default: Weights[] = 1.
  weights A double array of size x.GetLength(0) containing the weight for each observation.

Description

ClusterKMeans is an implementation of Algorithm AS 136 by Hartigan and Wong (1979). It computes $K$-means (centroid) Euclidean metric clusters for an input matrix starting with initial estimates of the $K$ cluster means. It allows for missing values (coded as NaN, not a number) and for weights and frequencies.

Let $p$ denote the number of variables to be used in computing the Euclidean distance between observations. The idea in $K$-means cluster analysis is to find a clustering (or grouping) of the observations so as to minimize the total within-cluster sums of squares. In this case, the total
sums of squares within each cluster is computed as the sum of the centered sum of squares over all nonmissing values of each variable. That is,

$$\phi = \sum_{i=1}^{K} \sum_{m=1}^{p} n_i \sum_{j=1}^{p} f_{\nu_{im}} w_{\nu_{im}} \delta_{\nu_{im},j} (x_{\nu_{im},j} - \bar{x}_{ij})^2$$

where $\nu_{im}$ denotes the row index of the $m$-th observation in the $i$-th cluster in the matrix $X$; $n_i$ is the number of rows of $X$ assigned to group $i$; $f$ denotes the frequency of the observation; $w$ denotes its weight; $d$ is zero if the $j$-th variable on observation $\nu_{im}$ is missing, otherwise $\delta$ is one; and $\bar{x}_{ij}$ is the average of the nonmissing observations for variable $j$ in group $i$. This method sequentially processes each observation and reassigns it to another cluster if doing so results in a decrease in the total within-cluster sums of squares. See Hartigan and Wong (1979) or Hartigan (1975) for details.

**Example: K-means Cluster Analysis**

This example performs K-means cluster analysis on Fisher’s iris data. The initial cluster seed for each iris type is an observation known to be in the iris type.

```c
/*
* ----------------------------------------------------------------------
* Copyright (c) 1999 Visual Numerics Inc. All Rights Reserved.
* ...
* VISUAL NUMERICS MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE
* SUITABILITY OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING
* BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. VISUAL
* NUMERICS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE
* AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR
* ITS DERIVATIVES.
* ----------------------------------------------------------------------
*/
using System;
using Imsl.Stat;
using Imsl.Math;

class ClusterKMeansEx1
{
    public static void Main(String[] argv)
    {
        double[, ] x = {{{5.100, 3.500, 1.400, 0.200},
            {4.900, 3.000, 1.400, 0.200},
            {4.700, 3.200, 1.300, 0.200},
            {4.600, 3.100, 1.500, 0.200},
            {5.000, 3.600, 1.400, 0.200}},
```
{5.900, 3.000, 4.200, 1.500},
{6.000, 2.200, 4.000, 1.000},
{6.100, 2.900, 4.700, 1.400},
{5.600, 2.900, 3.600, 1.300},
{6.700, 3.100, 4.400, 1.400},
{5.600, 3.000, 4.500, 1.500},
{5.800, 2.700, 4.100, 1.000},
{6.200, 2.200, 4.500, 1.500},
{5.600, 2.500, 3.900, 1.100},
{5.900, 3.200, 4.800, 1.800},
{6.100, 2.800, 4.000, 1.300},
{6.300, 2.500, 4.900, 1.500},
{6.100, 2.800, 4.700, 1.200},
{6.400, 2.900, 4.300, 1.300},
{6.600, 3.000, 4.400, 1.400},
{6.800, 2.800, 4.800, 1.400},
{6.700, 3.000, 5.000, 1.700},
{6.000, 2.900, 4.500, 1.500},
{5.700, 2.600, 3.500, 1.000},
{5.500, 2.400, 3.800, 1.100},
{5.500, 2.400, 3.700, 1.000},
{5.800, 2.700, 3.900, 1.200},
{6.000, 2.700, 5.100, 1.600},
{5.400, 3.000, 4.500, 1.500},
{6.000, 3.400, 4.500, 1.600},
{6.700, 3.100, 4.700, 1.500},
{6.300, 2.300, 4.400, 1.300},
{5.600, 3.000, 4.100, 1.300},
{5.500, 2.500, 4.000, 1.300},
{5.500, 2.600, 4.400, 1.200},
{6.100, 3.000, 4.600, 1.400},
{5.800, 2.600, 4.000, 1.200},
{5.000, 2.300, 3.300, 1.000},
{5.600, 2.700, 4.200, 1.300},
{5.700, 3.000, 4.200, 1.200},
{5.700, 2.900, 4.200, 1.300},
{6.200, 2.900, 4.300, 1.300},
{5.100, 2.500, 3.000, 1.100},
{5.700, 2.800, 4.100, 1.300},
{6.300, 3.300, 6.000, 2.500},
{5.800, 2.700, 5.100, 1.900},
{7.100, 3.000, 5.900, 2.100},
{6.300, 2.900, 5.600, 1.800},
{6.500, 3.000, 5.800, 2.000},
{7.600, 3.000, 6.600, 2.100},
{4.900, 2.500, 4.500, 1.700},
{7.300, 2.900, 6.300, 1.800},
{6.700, 2.500, 5.800, 1.800},
{7.200, 3.600, 6.100, 2.500},
{6.500, 3.200, 5.100, 2.000},
{6.400, 2.700, 5.300, 1.900},
{6.800, 3.000, 5.500, 2.100},
{5.700, 2.500, 5.000, 2.000},
{5.800, 2.800, 5.100, 2.400},
{6.400, 3.200, 5.300, 2.300},
{6.500, 3.000, 5.500, 1.800},

Multivariate Analysis

ClusterKMeans Class • 491
ClusterKMeans kmean = new ClusterKMeans(x, cs);

double[,] cm = kmean.Compute();
double[] wss = kmean.GetClusterSSQ();
int[] ic = kmean.GetClusterMembership();
int[] nc = kmean.GetClusterCounts();

PrintMatrix pm = new PrintMatrix("Cluster Means");

PrintMatrixFormat pmf = new PrintMatrixFormat();

pmf.NumberFormat = "0.0000";

pm.Print(pmf, cm);

new PrintMatrix("Cluster Membership").Print(ic);
new PrintMatrix("Sum of Squares").Print(wss);
new PrintMatrix("Number of observations").Print(nc);
Output

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.0060 3.4280 1.4620 0.2460</td>
</tr>
<tr>
<td>1</td>
<td>5.9016 2.7484 4.3935 1.4339</td>
</tr>
<tr>
<td>2</td>
<td>6.8500 3.0737 5.7421 2.0711</td>
</tr>
</tbody>
</table>

Cluster Membership

0
0 1
1 1
2 1
3 1
4 1
5 1
6 1
7 1
8 1
9 1
10 1
11 1
12 1
13 1
14 1
15 1
16 1
17 1
18 1
19 1
20 1
21 1
22 1
23 1
24 1
25 1
26 1
27 1
28 1
29 1
30 1
31 1
32 1
33 1
34 1
35 1
36 1
37 1
38 1
39 1

Multivariate Analysis
<table>
<thead>
<tr>
<th>Sum of Squares</th>
</tr>
</thead>
</table>

**Multivariate Analysis**  
ClusterKMeans Class • 495
Dissimilarities Class

Computes a matrix of dissimilarities (or similarities) between the columns (or rows) of a matrix.

```csharp
public class Imsl.Stat.Dissimilarities

Property

DistanceMatrix

virtual public double[,] DistanceMatrix {get; }

The distance matrix.

Constructors

Dissimilarities

public Dissimilarities(double[,] x, int distanceMethod, int distanceScale, int iRow)

Constructor for Dissimilarities.

Acceptable values of distanceMethod are 1, 2, ..., 8. See Remarks section of the Dissimilarities documentation for a description of these methods.

<table>
<thead>
<tr>
<th>distanceScale</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No scaling is performed.</td>
</tr>
<tr>
<td>1</td>
<td>Scale each column (row if iRow=1) by the standard deviation of the column(row).</td>
</tr>
<tr>
<td>2</td>
<td>Scale each column (row if iRow=1) by the range of the column (row).</td>
</tr>
</tbody>
</table>

If iRow = 1, distances are computed between the rows of x. Otherwise, distances between the columns of x are computed.
x A double matrix containing the data input matrix.
distanceMethod An int identifying the method to use in computing the dissimilarities or similarities.
distanceScale An int containing the scaling option.
iRow An int identifying whether distances are computed between rows or columns of x.

Imsl.Stat.ScaleFactorZeroException is thrown when computations cannot continue because a scale factor is zero.

Imsl.Stat.NoPositiveVarianceException is thrown when no variable has positive variance.

Imsl.Stat.ZeroNormException is thrown when the Euclidean norm of a column is equal to zero.

Dissimilarities

public Dissimilarities(double[,] x, int distanceMethod, int distanceScale, int iRow, int[] indexArray)
Constructor for Dissimilarities.

Acceptable values of distanceMethod are 1, 2, ..., 8. See Remarks section of the Dissimilarities documentation for a description of these methods.

<table>
<thead>
<tr>
<th>distanceScale</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No scaling is performed.</td>
</tr>
<tr>
<td>1</td>
<td>Scale each column (row if iRow=1) by the standard deviation of the column(row).</td>
</tr>
<tr>
<td>2</td>
<td>Scale each column (row if iRow=1) by the range of the column (row).</td>
</tr>
</tbody>
</table>

If iRow = 1, distances are computed between the rows of x. Otherwise, distances between the columns of x are computed.

x A double matrix containing the data input matrix.
distanceMethod An int identifying the method to use in computing the dissimilarities or similarities.
distanceScale An int containing the scaling option.
iRow An int identifying whether distances are computed between rows or columns of x.
indexArray An int array containing the indices of the rows (columns if iRow is 1) to use in computing the distance measure.

Imsl.Stat.ScaleFactorZeroException is thrown when computations cannot continue because a scale factor is zero.

Imsl.Stat.NoPositiveVarianceException is thrown when no variable has positive variance.

Imsl.Stat.ZeroNormException is thrown when the Euclidean norm of a column is equal to zero.
Description

Class Dissimilarities computes an upper triangular matrix (excluding the diagonal) of dissimilarities (or similarities) between the columns or rows of a matrix. Nine different distance measures can be computed. For the first three measures, three different scaling options can be employed. The distance matrix computed is generally used as input to clustering or multidimensional scaling functions.

The following discussion assumes that the distance measure is being computed between the columns of the matrix. If distances between the rows of the matrix are desired, set iRow to 1 when calling the Dissimilarities constructor.

For distanceMethod = 0 to 2, each row of x is first scaled according to the value of distanceScale. The scaling parameters are obtained from the values in the row scaled as either the standard deviation of the row or the row range; the standard deviation is computed from the unbiased estimate of the variance. If distanceScale is 0, no scaling is performed, and the parameters in the following discussion are all 1.0. Once the scaling value (if any) has been computed, the distance between column i and column j is computed via the difference vector $z_k = (x_k - y_k)s_k,i = 1, \ldots, ndstm$, where $x_k$ denotes the k-th element in the i-th column, $y_k$ denotes the corresponding element in the j-th column, and ndstm is the number of rows if differencing columns and the number of columns if differencing rows. For given $z_i$, the metrics 0 to 2 are defined as:

<table>
<thead>
<tr>
<th>distanceMethod</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Euclidean distance ($L_2$ norm)</td>
</tr>
<tr>
<td>1</td>
<td>Sum of the absolute differences ($L_1$ norm)</td>
</tr>
<tr>
<td>2</td>
<td>Maximum difference ($L_\infty$ norm)</td>
</tr>
</tbody>
</table>

Distance measures corresponding to distanceMethod = 3 to 8 do not allow for scaling.

<table>
<thead>
<tr>
<th>distanceMethod</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Mahalanobis distance</td>
</tr>
<tr>
<td>4</td>
<td>Absolute value of the cosine of the angle between the vectors</td>
</tr>
<tr>
<td>5</td>
<td>Angle in radians (0, pi) between the lines through the origin defined by the vectors</td>
</tr>
<tr>
<td>6</td>
<td>Correlation coefficient</td>
</tr>
<tr>
<td>7</td>
<td>Absolute value of the correlation coefficient</td>
</tr>
<tr>
<td>8</td>
<td>Number of exact matches, where $x_i = y_i$.</td>
</tr>
</tbody>
</table>

For the Mahalanobis distance, any variable used in computing the distance measure that is (numerically) linearly dependent upon the previous variables in the indexArray vector is omitted from the distance measure.

Example: Dissimilarities

The following example illustrates the use of Dissimilarities for computing the Euclidean distance between the rows of a matrix:

498 • Dissimilarities Class

IMSL C# Numerical Library
using System;
using Imsl.Math;
using Imsl.Stat;

public class DissimilaritiesEx1
{
    public static void Main(String[] args)
    {
        double[,] x = {{1.0, 1.0}, {1.0, 0.0}, {1.0, -1.0}, {1.0, 2.0}};
        int distanceMethod = 0;
        int distanceScale = 0;
        int iRow = 1;
        Dissimilarities dist = new Dissimilarities(x, distanceMethod, distanceScale, iRow);
        new PrintMatrix("Distance Matrix").Print(dist.DistanceMatrix);
    }
}

Output
Distance Matrix
    0 1 2 3
0 0 1 2 1
1 0 0 1 2
2 0 0 0 3
3 0 0 0 0

ClusterHierarchical Class

Performs a hierarchical cluster analysis from a distance matrix.

public class Imsl.Stat.ClusterHierarchical

Properties

ClusterLeftSons
    virtual public int[] ClusterLeftSons {get; }
    The left sons of each merged cluster. An int array containing the left sons of each merged cluster.

ClusterLevel
    virtual public double[] ClusterLevel {get; }
    The level at which the clusters are joined. A double array containing the level at which the clusters are joined.
Element \([k-1]\) contains the distance (or similarity) level at which cluster \(npt + k\) was formed. If the original data in \(dist\) was transformed, the inverse transformation is applied to the returned values.

ClusterRightSons

```csharp
virtual public int[] ClusterRightSons {get; }
```

The right sons of each merged cluster. An `int` array containing the right sons of each merged cluster.

### Constructor

```csharp
public ClusterHierarchical(double[,] dist, int method, int transform)
```

Constructor for `ClusterHierarchical`.

On input, only the upper triangular part of `dist` needs to be present.

**ClusterHierarchical** saves the upper triangular part in the lower triangle. On return, the upper triangular part of `dist` is restored, and the matrix is made symmetric.

<table>
<thead>
<tr>
<th>method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Single linkage (minimum distance).</td>
</tr>
<tr>
<td>1</td>
<td>Complete linkage (maximum distance).</td>
</tr>
<tr>
<td>2</td>
<td>Average distance within (average distance between objects within the merged cluster).</td>
</tr>
<tr>
<td>3</td>
<td>Average distance between (average distance between objects in the two clusters).</td>
</tr>
<tr>
<td>4</td>
<td>Ward’s method (minimize the within-cluster sums of squares). For Ward’s method, the elements of <code>dist</code> are assumed to be Euclidean distances.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>transform</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No transformation is required. The elements of <code>dist</code> are distances.</td>
</tr>
<tr>
<td>1</td>
<td>Convert similarities to distances by multiplying -1.0.</td>
</tr>
<tr>
<td>2</td>
<td>Convert similarities (usually correlations) to distances by taking the reciprocal of the absolute value.</td>
</tr>
</tbody>
</table>

`dist`  A `double` symmetric matrix containing the distance (or similarity) matrix.

`method`  An `int` identifying the clustering method to use.

`transform`  An `int` identifying the type of transformation applied to the measures in `dist`.

### Methods

**GetClusterMembership**

```csharp
public int[] GetClusterMembership(int nClusters)
```

Returns the cluster membership of each observation.
**nClusters**  An int which specifies the desired number of clusters.

Returns — An int array containing the cluster membership of each observation.

GetObsPerCluster

```java
public int[] GetObsPerCluster(int nClusters)
```

Returns the number of observations in each cluster.

**nClusters**  An int which specifies the desired number of clusters.

Returns — An int array containing the number of observations in each cluster.

### Description

Class **ClusterHierarchical** conducts a hierarchical cluster analysis based upon a distance matrix, or by appropriate use of the argument `transform`, based upon a similarity matrix. Only the upper triangular part of the `dist` matrix is required as input.

Hierarchical clustering in **ClusterHierarchical** proceeds as follows:

- Initially, each data point is considered to be a cluster, numbered 1 to \( n = npt \), where \( npt \) is the number of rows in `dist`.
- If the data matrix contains similarities, they are converted to distances by the method specified by the argument `transform`. Set \( k = 1 \).
- A search is made of the distance matrix to find the two closest clusters. These clusters are merged to form a new cluster, numbered \( n + k \). The cluster numbers of the two clusters joined at this stage are saved as `Right Sons` and `Left Sons`, and the distance measure between the two clusters is stored as `Cluster Level`.
- Based upon the method of clustering, updating of the distance measure in the row and column of `dist` corresponding to the new cluster is performed.
- Set \( k = k + 1 \). If \( k \) is less than \( n \), go to Step 2.

The five methods differ primarily in how the distance matrix is updated after two clusters have been joined. The argument `method` specifies how the distance of the cluster just merged with each of the remaining clusters will be updated. Class **ClusterHierarchical** allows five methods for computing the distances. To understand these measures, suppose in the following discussion that clusters \( A \) and \( B \) have just been joined to form cluster \( Z \), and interest is in computing the distance of \( Z \) with another cluster called \( C \).
<table>
<thead>
<tr>
<th>method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Single linkage (minimum distance). The distance from ( Z ) to ( C ) is the minimum of the distances ( A ) to ( C ), ( B ) to ( C ).</td>
</tr>
<tr>
<td>1</td>
<td>Complete linkage (maximum distance). The distance from ( Z ) to ( C ) is the maximum of the distances ( A ) to ( C ), ( B ) to ( C ).</td>
</tr>
<tr>
<td>2</td>
<td>Average-distance-within-clusters method. The distance from ( Z ) to ( C ) is the average distance of all objects that would be within the cluster formed by merging clusters ( Z ) and ( C ). This average may be computed according to formulas given by Anderberg (1973, page 139).</td>
</tr>
<tr>
<td>3</td>
<td>Average-distance-between-clusters method. The distance from ( Z ) to ( C ) is the average distance of objects within cluster ( Z ) to objects within cluster ( C ). This average may be computed according to methods given by Anderberg (1973, page 140).</td>
</tr>
<tr>
<td>4</td>
<td>Ward’s method: Clusters are formed so as to minimize the increase in the within-cluster sums of squares. The distance between two clusters is the increase in these sums of squares if the two clusters were merged. A method for computing this distance from a squared Euclidean distance matrix is given by Anderberg (1973, pages 142-145).</td>
</tr>
</tbody>
</table>

In general, single linkage will yield long thin clusters while complete linkage will yield clusters that are more spherical. Average linkage and Ward’s linkage tend to yield clusters that are similar to those obtained with complete linkage.

Class `ClusterHierarchical` produces a unique representation of the binary cluster tree via the following three conventions; the fact that the tree is unique should aid in interpreting the clusters. First, when two clusters are joined and each cluster contains two or more data points, the cluster initially formed with the smallest level becomes the left son. Second, when a cluster containing more than one data point is joined with a cluster containing a single data point, the cluster with the single data point becomes the right son. Third, when two clusters containing
only one object are joined, the cluster with the smallest cluster number becomes the right son.

Comments

• The clusters corresponding to the original data points are numbered from 1 to \( npt \), where \( npt \) is the number of rows in \( dist \). The \( npt - 1 \) clusters formed by merging clusters are numbered \( npt + 1 \) to \( npt + (npt - 1) \).

• Raw correlations, if used as similarities, should be made positive and transformed to a distance measure. One such transformation can be performed by setting argument \( transform \), with \( transform = 2 \).

• The user may cluster either variables or observations with \texttt{ClusterHierarchical} since a dissimilarity matrix, not the original data, is used. Class \texttt{Imsl.Stat.Dissimilarities} (p. 496) may be used to compute the matrix \( dist \) for either the variables or observations.

Example: ClusterHierarchical

This example illustrates a typical usage of \texttt{ClusterHierarchical}. The Fisher iris data is clustered. First the distance between irises is computed using the class \texttt{Dissimilarities}. The resulting distance matrix is then clustered using \texttt{ClusterHierarchical}, and cluster memberships for 5 clusters are computed.

```csharp
using System;
using Imsl.Math;
using Imsl.Stat;

public class ClusterHierarchicalEx1
{
    public static void Main(String[] args)
    {
        double[,] irisData = {
            { 5.1, 3.5, 1.4, .2},
            { 4.9, 3.0, 1.4, .2},
            { 4.7, 3.2, 1.3, .2},
            { 4.6, 3.1, 1.5, .2},
            { 5.0, 3.6, 1.4, .2},
            { 5.4, 3.9, 1.7, .4},
            { 4.6, 3.4, 1.4, .3},
            { 5.0, 3.4, 1.5, .2},
            { 4.4, 2.9, 1.4, .2},
            { 4.9, 3.1, 1.5, .1},
            { 5.4, 3.7, 1.5, .2},
            { 4.8, 3.4, 1.6, .2},
            { 4.8, 3.0, 1.4, .1},
            { 4.3, 3.0, 1.1, .1},
            { 5.8, 4.0, 1.2, .2},
            { 5.7, 4.4, 1.5, .4},
            { 5.4, 3.9, 1.3, .4},
            { 5.1, 3.5, 1.4, .3},
        }
```
{ 5.7, 3.8, 1.7, .3},
{ 5.1, 3.8, 1.5, .3},
{ 5.4, 3.4, 1.7, .2},
{ 5.1, 3.7, 1.5, .4},
{ 4.6, 3.6, 1.0, .2},
{ 5.1, 3.3, 1.7, .5},
{ 4.8, 3.4, 1.9, .2},
{ 5.0, 3.0, 1.6, .2},
{ 5.0, 3.4, 1.6, .4},
{ 5.2, 3.5, 1.5, .2},
{ 5.2, 3.4, 1.4, .2},
{ 4.7, 3.2, 1.6, .2},
{ 4.8, 3.1, 1.6, .2},
{ 5.4, 3.4, 1.5, .4},
{ 5.2, 4.1, 1.5, .1},
{ 5.5, 4.2, 1.4, .2},
{ 4.9, 3.1, 1.5, .2},
{ 5.0, 3.2, 1.2, .2},
{ 5.5, 3.5, 1.3, .2},
{ 4.9, 3.6, 1.4, .1},
{ 4.4, 3.0, 1.3, .2},
{ 5.1, 3.4, 1.5, .2},
{ 5.0, 3.5, 1.3, .3},
{ 4.5, 2.3, 1.3, .3},
{ 4.4, 3.2, 1.3, .2},
{ 5.0, 3.5, 1.6, .6},
{ 5.1, 3.8, 1.9, .4},
{ 4.8, 3.0, 1.4, .3},
{ 5.1, 3.8, 1.6, .2},
{ 4.6, 3.2, 1.4, .2},
{ 5.3, 3.7, 1.5, .2},
{ 5.0, 3.3, 1.4, .2},
{ 7.0, 3.2, 4.7, 1.4},
{ 6.4, 3.2, 4.5, 1.5},
{ 6.9, 3.1, 4.9, 1.5},
{ 5.5, 2.3, 4.0, 1.3},
{ 6.5, 2.8, 4.6, 1.5},
{ 5.7, 2.8, 4.5, 1.3},
{ 6.3, 3.3, 4.7, 1.6},
{ 4.9, 2.4, 3.3, 1.0},
{ 6.6, 2.9, 4.6, 1.3},
{ 5.2, 2.7, 3.9, 1.4},
{ 5.0, 2.0, 3.5, 1.0},
{ 5.9, 3.0, 4.2, 1.5},
{ 6.0, 2.2, 4.0, 1.0},
{ 6.1, 2.9, 4.7, 1.4},
{ 5.6, 2.9, 3.6, 1.3},
{ 6.7, 3.1, 4.4, 1.4},
{ 5.6, 3.0, 4.5, 1.5},
{ 5.8, 2.7, 4.1, 1.0},
{ 6.2, 2.2, 4.5, 1.5},
{ 5.6, 2.5, 3.9, 1.1},
{ 5.9, 3.2, 4.8, 1.8},
{ 6.1, 2.8, 4.0, 1.3},
{ 6.3, 2.5, 4.9, 1.5},
{ 6.1, 2.8, 4.7, 1.2},

504 • ClusterHierarchical Class  IMSL C# Numerical Library
\{6.4, 2.9, 4.3, 1.3\},
\{6.6, 3.0, 4.4, 1.4\},
\{6.8, 2.8, 4.8, 1.4\},
\{6.7, 3.0, 5.0, 1.7\},
\{6.0, 2.9, 4.5, 1.5\},
\{5.7, 2.6, 3.5, 1.0\},
\{5.5, 2.4, 3.8, 1.1\},
\{5.5, 2.4, 3.7, 1.0\},
\{5.8, 2.7, 3.9, 1.2\},
\{6.0, 2.7, 5.1, 1.6\},
\{5.4, 3.0, 4.5, 1.5\},
\{6.0, 3.4, 4.5, 1.6\},
\{6.7, 3.1, 4.7, 1.5\},
\{6.3, 2.3, 4.4, 1.3\},
\{5.6, 3.0, 4.1, 1.3\},
\{5.5, 2.5, 4.0, 1.3\},
\{5.5, 2.6, 4.4, 1.2\},
\{6.1, 3.0, 4.6, 1.4\},
\{5.8, 2.6, 4.0, 1.2\},
\{5.0, 2.3, 3.3, 1.0\},
\{5.6, 2.7, 4.2, 1.3\},
\{5.7, 3.0, 4.2, 1.2\},
\{5.7, 2.9, 4.2, 1.3\},
\{6.2, 2.9, 4.3, 1.3\},
\{5.1, 2.5, 3.0, 1.1\},
\{5.7, 2.8, 4.1, 1.3\},
\{6.3, 3.3, 6.0, 2.5\},
\{5.8, 2.7, 5.1, 1.9\},
\{7.1, 3.0, 5.9, 2.1\},
\{6.3, 2.9, 5.6, 1.8\},
\{6.5, 3.0, 5.8, 2.2\},
\{7.6, 3.0, 6.6, 2.1\},
\{4.9, 2.5, 4.5, 1.7\},
\{7.3, 2.9, 6.3, 1.8\},
\{6.7, 2.5, 5.8, 1.8\},
\{7.2, 3.6, 6.1, 2.5\},
\{6.5, 3.2, 5.1, 2.0\},
\{6.4, 2.7, 5.3, 1.9\},
\{6.8, 3.0, 5.5, 2.1\},
\{5.7, 2.5, 5.0, 2.0\},
\{5.8, 2.8, 5.1, 2.4\},
\{6.4, 3.2, 5.3, 2.3\},
\{6.5, 3.0, 5.5, 1.8\},
\{7.7, 3.8, 6.7, 2.2\},
\{7.7, 2.6, 6.9, 2.3\},
\{6.0, 2.2, 5.0, 1.5\},
\{6.9, 3.2, 5.7, 2.3\},
\{5.6, 2.8, 4.9, 2.0\},
\{7.7, 2.8, 6.7, 2.0\},
\{6.3, 2.7, 4.9, 1.8\},
\{6.7, 3.3, 5.7, 2.1\},
\{7.2, 3.2, 6.0, 1.8\},
\{6.2, 2.8, 4.8, 1.8\},
\{6.1, 3.0, 4.9, 1.8\},
\{6.4, 2.8, 5.6, 2.1\},
\{7.2, 3.0, 5.8, 1.6\}.
{ 7.4, 2.8, 6.1, 1.9},
{ 7.9, 3.8, 6.4, 2.0},
{ 6.4, 2.8, 5.6, 2.2},
{ 6.3, 2.8, 5.1, 1.5},
{ 6.1, 2.6, 5.6, 1.4},
{ 7.7, 3.0, 6.1, 2.3},
{ 6.3, 3.4, 5.6, 2.4},
{ 6.4, 3.1, 5.5, 1.8},
{ 6.0, 3.0, 4.8, 1.8},
{ 6.9, 3.1, 5.4, 2.1},
{ 6.7, 3.1, 5.6, 2.4},
{ 6.9, 3.1, 5.1, 2.3},
{ 5.8, 2.7, 5.1, 1.9},
{ 6.8, 3.2, 5.9, 2.3},
{ 6.7, 3.3, 5.7, 2.5},
{ 6.7, 3.0, 5.2, 2.3},
{ 6.3, 2.5, 5.0, 1.9},
{ 6.5, 3.0, 5.2, 2.0},
{ 6.2, 3.4, 5.4, 2.3},
{ 5.9, 3.0, 5.1, 1.8});

Dissimilarities dist = new Dissimilarities(irisData, 0, 1, 1);
ClusterHierarchical clink = new ClusterHierarchical(dist.DistanceMatrix, 2, 0);

int nClusters = 5;
int[] iclus = clink.GetClusterMembership(nClusters);
int[] nclus = clink.GetObsPerCluster(nClusters);
System.Console.Out.WriteLine("Cluster Membership");
for (int i = 0; i < 15; i++)
{
    for (int j = 0; j < 10; j++)
        Console.Out.Write(iclus[i * 10 + j] + " ");
    Console.Out.WriteLine();
}

for (int i = 0; i < nClusters; i++)
    System.Console.Out.Write(nclus[i] + " ");
System.Console.Out.WriteLine();
}

Output

Cluster Membership
5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5
3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3

506 • ClusterHierarchical Class
IMSL C# Numerical Library
**FactorAnalysis Class**

Performs Principal Component Analysis or Factor Analysis on a covariance or correlation matrix.

```java
public class Imsl.Stat.FactorAnalysis
```

**Properties**

`ConvergenceCriterion1`

```java
public double ConvergenceCriterion1 {get; set; }
```

The convergence criterion used to terminate the iterations. A `double` used to terminate the iterations.

For the least squares and and maximum likelihood methods convergence is assumed when the relative change in the criterion is less than `ConvergenceCriterion1`. For alpha factor analysis, convergence is assumed when the maximum change (relative to the variance) of a uniqueness is less than `ConvergenceCriterion1`. `ConvergenceCriterion1` is not referenced for the other estimation methods. By default, `ConvergenceCriterion1` is set to 0.0001.

`ConvergenceCriterion2`

```java
public double ConvergenceCriterion2 {get; set; }
```

The convergence criterion used to switch to exact second derivatives. A `double` used to switch to exact second derivatives.

When the largest relative change in the unique standard deviation vector is less than `ConvergenceCriterion2`, exact second derivative vectors are used. By default, `ConvergenceCriterion2` is set to 0.1. Not referenced for principal component, principal factor, image factor, or alpha factor methods.

`DegreesOfFreedom`

```java
public int DegreesOfFreedom {get; set; }
```

The number of degrees of freedom. An `int` value specifying the number of degrees of freedom in the input matrix.

If this property is not set, 100 degrees of freedom are assumed.
FactorLoadingEstimationMethod

public Imsl.Stat.FactorAnalysis.Model FactorLoadingEstimationMethod {get; set; }

The factor loading estimation method. Indicates the method to be applied for obtaining the factor loadings. Use FactorAnalysis.Model field PrincipalComponent, PrincipalFactor, UnweightedLeastSquares, GeneralizedLeastSquares, MaximumLikelihood, ImageFactorAnalysis, or AlphaFactorAnalysis for FactorLoadingEstimationMethod. By default, the PrincipalComponent is used.

For the principal component and principal factor methods, the factor loading estimates are computed as

$$\hat{\Gamma} \hat{\Delta}^{-1/2}$$

where \(\Gamma\) and the diagonal matrix \(\Delta\) are the eigenvalues and eigenvectors of a matrix. In the principal component model, the eigensystem analysis is performed on the sample covariance (correlation) matrix \(S\) while in the principal factor model the matrix \((S - \Psi)\) is used. If the unique error variances \(\Psi\) are not known in the principal factor model, then they are estimated. This is achieved by setting the property VarianceEstimationMethod to 0. If the principal component model is used, the error variances in the Variances property are set to 0.0 automatically.

The basic idea in the principal component method is to find factors that maximize the variance in the original data that is explained by the factors. Because this method allows the unique errors to be correlated, some factor analysts insist that the principal component method is not a factor analytic method. Usually however, the estimates obtained via the principal component model and other models in factor analysis will be quite similar.

It should be noted that both the principal component and the principal factor methods give different results when the correlation matrix is used in place of the covariance matrix. Indeed, any rescaling of the sample covariance matrix can lead to different estimates with either of these methods. A further difficulty with the principal factor method is the problem of estimating the unique error variances. Theoretically, these must be known in advance and set using the the Variances property. In practice, the estimates of these parameters produced by setting the property VarianceEstimationMethod to 0 are often used. In either case, the resulting adjusted covariance (correlation) matrix

$$(S - \hat{\Psi})$$

may not yield the \(nfactors\) positive eigenvalues required for \(nfactors\) factors to be obtained. If this occurs, the user must either lower the number of factors to be estimated or give new unique error variance values.

For the least-squares and maximum likelihood methods an iterative algorithm is used to obtain the estimates (see joreskog 1977). As with the principal factor model, the user may either input the initial unique error variances or allow the algorithm to compute initial estimates. Unlike the principal factor method, the code then optimizes the criterion function with respect to both \(\Psi\) and \(\Gamma\). (In the principal factor method, \(\Psi\) is assumed to be known. Given \(\Psi\), estimates for \(\Lambda\) may be obtained.)

The major differences between the estimation methods described in this member function are in the criterion function that is optimized. Let \(S\) denote the sample covariance
(correlation) matrix, and let \( \Sigma \) denote the covariance matrix that is to be estimated by the factor model. In the unweighted least-squares method, also called the iterated principal factor method or the minres method (see Harman 1976, page 177), the function minimized is the sum of the squared differences between \( S \) and \( \Sigma \). This is written as \( \Phi_{ul} = \text{trace}((S - \Sigma)^2) \).

Generalized least-squares and maximum likelihood estimates are asymptotically equivalent methods. Maximum likelihood estimates maximize the (normal theory) likelihood \( \{ \Phi_{ml} = \text{trace}(\Sigma^{-1}S) - \log(|\Sigma^{-1}S|) \} \), while generalized least squares optimizes the function \( \Phi_{gs} = \text{trace}(\Sigma S^{-1} - I)^2 \).

In all three methods, a two-stage optimization procedure is used. This proceeds by first solving the likelihood equations for \( \Lambda \) in terms of \( \Psi \) and substituting the solution into the likelihood. This gives a criterion \( \Phi(\Psi, \Lambda(\Psi)) \), which is optimized with respect to \( \Psi \). In the second stage, the estimates \( \hat{\Lambda} \) are obtained from the estimates for \( \Psi \).

The generalized least-squares and the maximum likelihood methods allow for the computation of a statistic for testing that \text{mfactors} common factors are adequate to fit the model. This is a chi-squared test that all remaining parameters associated with additional factors are zero. If the probability of a larger chi-squared is small (see \text{stat}[4]) so that the null hypothesis is rejected, then additional factors are needed (although these factors may not be of any practical importance). Failure to reject does not legitimize the model. The statistic \text{stat}[2] is a likelihood ratio statistic in maximum likelihood estimates. As such, it asymptotically follows a chi-squared distribution with degrees of freedom given in \text{stat}[3].

The Tucker and Lewis (1973) reliability coefficient, \( \rho \), is returned in \text{stat}[1] when the maximum likelihood or generalized least-squares methods are used. This coefficient is an estimate of the ratio of explained to the total variation in the data. It is computed as follows:

\[
\rho = \frac{mM_o - mM_k}{mM_o - 1}
\]

\[
m = d - \frac{2p + 5}{6} - \frac{2k}{6}
\]

\[
M_o = \frac{-\ln(|S|)}{p(p - 1)/2}
\]

\[
M_k = \frac{\Phi}{((p - k)^2 - p - k)/2}
\]

where \( |S| \) is the determinant of \text{cov}, \( p \) is the number of variables, \( k \) is the number of factors, \( \Phi \) is the optimized criterion, and \( d \) is the number of degrees of freedom.

The term “image analysis” is used here to denote the noniterative image method of Kaiser (1963). It is not the image factor analysis discussed by Harman (1976, page 226). The image method (as well as the alpha factor analysis method) begins with the notion that only a finite number from an infinite number of possible variables have been measured. The image factor pattern is calculated under the assumption that the ratio of
the number of factors to the number of observed variables is near zero so that a very good estimate for the unique error variances (for standardized variables) is given as one minus the squared multiple correlation of the variable under consideration with all variables in the covariance matrix.

First, the matrix $D^2 = (\text{diag}(S^{-1}))^{-1}$ is computed where the operator "diag" results in a matrix consisting of the diagonal elements of its argument, and $S$ is the sample covariance (correlation) matrix. Then, the eigenvalues $\Lambda$ and eigenvectors $\Gamma$ of the matrix $D^{-1}SD^{-1}$ are computed. Finally, the unrotated image factor pattern matrix is computed as $A = D\Gamma[(\Lambda - I)^2\Lambda^{-1}]^{1/2}$.

The alpha factor analysis method of Kaiser and Caffrey (1965) finds factor-loading estimates to maximize the correlation between the factors and the complete universe of variables of interest. The basic idea in this method is as follows: only a finite number of variables out of a much larger set of possible variables is observed. The population factors are linearly related to this larger set while the observed factors are linearly related to the observed variables. Let $f$ denote the factors obtainable from a finite set of observed random variables, and let $\xi$ denote the factors obtainable from the universe of observable variables. Then, the alpha method attempts to find factor-loading estimates so as to maximize the correlation between $f$ and $\xi$. In order to obtain these estimates, the iterative algorithm of Kaiser and Caffrey (1965) is used.

MaxIterations

```csharp
public int MaxIterations {get; set; }
```

The maximum number of iterations in the iterative procedure. An int used as the maximum number of iterations allowed during the iterative portion of the algorithm.

By default, `MaxIterations` is set to 60. `MaxIterations` is not referenced for factor loading methods `PrincipalComponent`, `PrincipalFactor`, or `ImageFactorAnalysis`.

MaxStep

```csharp
public int MaxStep {get; set; }
```

The maximum number of step halvings allowed during an iteration. An int used as the maximum number of step halvings allowed during an iteration.

If this property is not set, `MaxStep` is set to 8. `MaxStep` is not referenced for `PrincipalComponent`, `PrincipalFactor`, `ImageFactorAnalysis`, or `AlphaFactorAnalysis` methods.

VarianceEstimationMethod

```csharp
public int VarianceEstimationMethod {get; set; }
```

The variance estimation method. An int used to designate the method to be applied for obtaining the initial estimates of the unique variances.

By default, `VarianceEstimationMethod` is set to 1.

<table>
<thead>
<tr>
<th>init</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial estimates are taken as the constant $1 - \text{nfactors} / (2*\text{nvar})$ divided by the diagonal elements of the inverse of input matrix <code>cov</code>.</td>
</tr>
<tr>
<td>1</td>
<td>Initial estimates are input by the user in vector <code>uniq</code>.</td>
</tr>
</tbody>
</table>
Note that when the factor loading estimation method is PrincipalComponent, the initial estimates in uniq are reset to 0.0.

Constructor

FactorAnalysis
public FactorAnalysis(double[,] cov, Imsl.Stat.FactorAnalysis.MatrixType matrixType, int nfactors)
Constructor for FactorAnalysis.
FactorAnalysis.matrixType can specify a VarianceCovariance or Correlation matrix.

If nfactors is not known in advance, several different values of nfactors should be used, and the most reasonable value kept in the final solution. Since, in practice, the non-iterative methods often lead to solutions which differ little from the iterative methods, it is usually suggested that a non-iterative method be used in the initial stages of the factor analysis, and that the iterative methods be used once issues such as the number of factors have been resolved.

cov A double matrix containing the covariance or correlation matrix.
matrixType An int scalar indicating the type of matrix that is input.
nfactors An int scalar indicating the number of factors in the model.

System.ArgumentException is thrown if x.GetLength(0), and x.GetLength(1) are equal to 0

Methods

GetCorrelations
public double[,] GetCorrelations() Returns the correlations of the principal components.

If a covariance matrix is input to the constructor, then the correlations are with the observed variables. Otherwise, the correlations are with the standardized (to a variance of 1.0) variables. Only valid for the Principal Components Model.

Returns — A double matrix containing the correlations of the principal components with the observed/standardized variables.

Imsl.Stat.RankException is thrown if the rank of the covariance matrix is less than the number of factors.

Imsl.Stat.NoDegreesOfFreedomException is thrown if there are no degrees of freedom for the significance testing.

Imsl.Stat.NotSemiDefiniteException is thrown if the Hessian matrix not semi-definite.
Imsl.Stat.NotPositiveSemiDefiniteException is thrown if the covariance matrix is not positive semi-definite.

Imsl.Stat.NotPositiveDefiniteException is thrown if the covariance matrix is not positive definite because a variable is linearly related to other variables.

Imsl.Stat.SingularException is thrown if the covariance matrix is singular.

Imsl.Stat.BadVarianceException is thrown if the input variance is not in the allowed range.

Imsl.Stat.EigenvalueException is thrown if an error occurred in calculating the eigenvalues of the adjusted (inverse) covariance matrix. Check the input covariance matrix.

Imsl.Stat.NonPositiveEigenvalueException is thrown if in alpha factor analysis an eigenvalue is not positive. As all eigenvalues corresponding to the factors must be positive, either the number of factors must be reduced, or new initial estimates for the unique variances must be given.

GetFactorLoadings

```csharp
public double[,] GetFactorLoadings()
```

Returns the unrotated factor loadings.

Returns — A double matrix containing the unrotated factor loadings.

Imsl.Stat.RankException is thrown if the rank of the covariance matrix is less than the number of factors.

Imsl.Stat.NoDegreesOfFreedomException is thrown if there are no degrees of freedom for the significance testing.

Imsl.Stat.NotSemiDefiniteException is thrown if the Hessian matrix not semi-definite.

Imsl.Stat.NotPositiveSemiDefiniteException is thrown if the covariance matrix is not positive semi-definite.

Imsl.Stat.NotPositiveDefiniteException is thrown if the covariance matrix is not positive definite because a variable is linearly related to other variables.

Imsl.Stat.SingularException is thrown if the covariance matrix is singular.

Imsl.Stat.BadVarianceException is thrown if the input variance is not in the allowed range.

Imsl.Stat.EigenvalueException is thrown if an error occurred in calculating the eigenvalues of the adjusted (inverse) covariance matrix. Check the input covariance matrix.

Imsl.Stat.NonPositiveEigenvalueException is thrown if in alpha factor analysis an eigenvalue is not positive. As all eigenvalues corresponding to the factors must be positive, either the number of factors must be reduced, or new initial estimates for the unique variances must be given.
GetParameterUpdates

```java
public double[] GetParameterUpdates()
```

Returns the parameter updates.

The parameter updates are only meaningful for the common factor model. The parameter updates are set to 0.0 for the principal component model.

Returns — A double array containing the parameter updates when convergence was reached (or the iterations terminated).

**Imsl.Stat.RankException** is thrown if the rank of the covariance matrix is less than the number of factors.

**Imsl.Stat.NoDegreesOfFreedomException** is thrown if there are no degrees of freedom for the significance testing.

**Imsl.Stat.NotSemiDefiniteException** is thrown if the Hessian matrix is not semi-definite.

**Imsl.Stat.NotPositiveSemiDefiniteException** is thrown if the covariance matrix is not positive semi-definite.

**Imsl.Stat.NotPositiveDefiniteException** is thrown if the covariance matrix is not positive definite because a variable is linearly related to other variables.

**Imsl.Stat.SingularException** is thrown if the covariance matrix is singular.

**Imsl.Stat.BadVarianceException** is thrown if the input variance is not in the allowed range.

**Imsl.Stat.EigenvalueException** is thrown if an error occurred in calculating the eigenvalues of the adjusted (inverse) covariance matrix. Check the input covariance matrix.

**Imsl.Stat.NonPositiveEigenvalueException** is thrown if in alpha factor analysis an eigenvalue is not positive. As all eigenvalues corresponding to the factors must be positive, either the number of factors must be reduced, or new initial estimates for the unique variances must be given.

GetPercents

```java
public double[] GetPercents()
```

Returns the cumulative percent of the total variance explained by each principal component.

Valid for the principal component model.

Returns — A double array containing the total variance explained by each principal component.

**Imsl.Stat.RankException** is thrown if the rank of the covariance matrix is less than the number of factors.

**Imsl.Stat.NoDegreesOfFreedomException** is thrown if there are no degrees of freedom for the significance testing.

**Imsl.Stat.NotSemiDefiniteException** is thrown if the Hessian matrix is not semi-definite.
Imsl.Stat.NotPositiveSemiDefiniteException is thrown if the covariance matrix is not positive semi-definite.

Imsl.Stat.NotPositiveDefiniteException is thrown if the covariance matrix is not positive definite because a variable is linearly related to other variables.

Imsl.Stat.SingularException is thrown if the covariance matrix is singular.

Imsl.Stat.BadVarianceException is thrown if the input variance is not in the allowed range.

Imsl.Stat.EigenvalueException is thrown if an error occurred in calculating the eigenvalues of the adjusted (inverse) covariance matrix. Check the input covariance matrix.

Imsl.Stat.NonPositiveEigenvalueException is thrown if in alpha factor analysis an eigenvalue is not positive. As all eigenvalues corresponding to the factors must be positive, either the number of factors must be reduced, or new initial estimates for the unique variances must be given.

GetStandardErrors
  public double[] GetStandardErrors()
  Returns the estimated asymptotic standard errors of the eigenvalues.
  Returns — A double array containing the estimated asymptotic standard errors of the eigenvalues.

Imsl.Stat.RankException is thrown if the rank of the covariance matrix is less than the number of factors.

Imsl.Stat.NoDegreesOfFreedomException is thrown if there are no degrees of freedom for the significance testing.

Imsl.Stat.NotSemiDefiniteException is thrown if the Hessian matrix is not semi-definite.

Imsl.Stat.NotPositiveSemiDefiniteException is thrown if the covariance matrix is not positive semi-definite.

Imsl.Stat.NotPositiveDefiniteException is thrown if the covariance matrix is not positive definite because a variable is linearly related to other variables.

Imsl.Stat.SingularException is thrown if the covariance matrix is singular.

Imsl.Stat.BadVarianceException is thrown if the input variance is not in the allowed range.

Imsl.Stat.EigenvalueException is thrown if an error occurred in calculating the eigenvalues of the adjusted (inverse) covariance matrix. Check the input covariance matrix.

Imsl.Stat.NonPositiveEigenvalueException is thrown if in alpha factor analysis an eigenvalue is not positive. As all eigenvalues corresponding to the factors must be positive, either the number of factors must be reduced, or new initial estimates for the unique variances must be given.
public double[] GetStatistics()
Returns statistics.

Statistics are not defined and set to NaN when the method used to obtain the estimates is the principal component method, principal factor method, image factor analysis method, or alpha analysis method.

<table>
<thead>
<tr>
<th>i</th>
<th>Statistics[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Value of the function minimum.</td>
</tr>
<tr>
<td>1</td>
<td>Tucker reliability coefficient.</td>
</tr>
<tr>
<td>2</td>
<td>Chi-squared test statistic for testing that the number of factors in the model are adequate for the data.</td>
</tr>
<tr>
<td>3</td>
<td>Degrees of freedom in chi-squared. This is computed as ((nvar - nfactors)^2 - nvar - nfactors)/2) where (nvar) is the number of variables and (nfactors) is the number of factors in the model.</td>
</tr>
<tr>
<td>4</td>
<td>Probability of a greater chi-squared statistic.</td>
</tr>
<tr>
<td>5</td>
<td>Number of iterations.</td>
</tr>
</tbody>
</table>

Returns — A double array containing output statistics.

**Imsl.Stat.RankException** is thrown if the rank of the covariance matrix is less than the number of factors.

**Imsl.Stat.NoDegreesOfFreedomException** is thrown if there are no degrees of freedom for the significance testing.

**Imsl.Stat.NotSemiDefiniteException** is thrown if the Hessian matrix not semi-definite.

**Imsl.Stat.NotPositiveSemiDefiniteException** is thrown if the covariance matrix is not positive semi-definite.

**Imsl.Stat.NotPositiveDefiniteException** is thrown if the covariance matrix is not positive definite because a variable is linearly related to other variables.

**Imsl.Stat.SingularException** is thrown if the covariance matrix is singular.

**Imsl.Stat.BadVarianceException** is thrown if the input variance is not in the allowed range.

**Imsl.Stat.EigenvalueException** is thrown if an error occurred in calculating the eigenvalues of the adjusted (inverse) covariance matrix. Check the input covariance matrix.

**Imsl.Stat.NonPositiveEigenvalueException** is thrown if in alpha factor analysis an eigenvalue is not positive. As all eigenvalues corresponding to the factors must be positive, either the number of factors must be reduced, or new initial estimates for the unique variances must be given.
GetValues
   public double[] GetValues()
   Returns the eigenvalues.

   If Alpha Factor analysis is used, then the first \texttt{nfactors} positions of the array contain the 
   Alpha coefficients. Here, \texttt{nfactors} is the number of factors in the model. If the algorithm 
   fails to converge for a particular eigenvalue, that eigenvalue is set to \texttt{NaN}. Note that the 
   eigenvalues are usually not the eigenvalues of the input matrix \texttt{cov}. They are the 
   eigenvalues of the input matrix \texttt{cov} when the Principal Component method is used.

   Returns — A \texttt{double} array containing the eigenvalues of the matrix from which the 
   factors were extracted ordered from largest to smallest.

   \texttt{Imsl.Stat.RankException} is thrown if the rank of the covariance matrix is less than the 
   number of factors.

   \texttt{Imsl.Stat.NoDegreesOfFreedomException} is thrown if there are no degrees of freedom 
   for the significance testing.

   \texttt{Imsl.Stat.NotSemiDefiniteException} is thrown if the Hessian matrix not 
   semi-definite.

   \texttt{Imsl.Stat.NotPositiveSemiDefiniteException} is thrown if the covariance matrix is 
   not positive semi-definite.

   \texttt{Imsl.Stat.NotPositiveDefiniteException} is thrown if the covariance matrix is 
   not positive definite because a variable is linearly related to other variables.

   \texttt{Imsl.Stat.SingularException} is thrown if the covariance matrix is singular.

   \texttt{Imsl.Stat.BadVarianceException} is thrown if the input variance is not in the allowed 
   range.

   \texttt{Imsl.Stat.EigenvalueException} is thrown if an error occurred in calculating the 
   eigenvalues of the adjusted (inverse) covariance matrix. Check the input covariance 
   matrix.

   \texttt{Imsl.Stat.NonPositiveEigenvalueException} is thrown if in alpha factor analysis an 
   eigenvalue is not positive. As all eigenvalues corresponding to the factors must be 
   positive, either the number of factors must be reduced, or new initial estimates for 
   the unique variances must be given.

GetVariances
   public double[] GetVariances()
   Returns the unique variances.

   Returns — A \texttt{double} array containing the unique variances.

   \texttt{Imsl.Stat.RankException} is thrown if the rank of the covariance matrix is less than the 
   number of factors.

   \texttt{Imsl.Stat.NoDegreesOfFreedomException} is thrown if there are no degrees of freedom 
   for the significance testing.

   \texttt{Imsl.Stat.NotSemiDefiniteException} is thrown if the Hessian matrix not 
   semi-definite.
Imsl.Stat.NotPositiveSemiDefiniteException is thrown if the covariance matrix is not positive semi-definite.

Imsl.Stat.NotPositiveDefiniteException is thrown if the covariance matrix is not positive definite because a variable is linearly related to other variables.

Imsl.Stat.SingularException is thrown if the covariance matrix is singular.

Imsl.Stat.BadVarianceException is thrown if the input variance is not in the allowed range.

Imsl.Stat.EigenvalueException is thrown if an error occurred in calculating the eigenvalues of the adjusted (inverse) covariance matrix. Check the input covariance matrix.

Imsl.Stat.NonPositiveEigenvalueException is thrown if in alpha factor analysis an eigenvalue is not positive. As all eigenvalues corresponding to the factors must be positive, either the number of factors must be reduced, or new initial estimates for the unique variances must be given.

GetVectors

public double[,] GetVectors() Returns the eigenvectors.

The j-th column of the eigenvector matrix corresponds to the j-th eigenvalue. The eigenvectors are normalized to each have Euclidean length equal to one. Also, the sign of each vector is set so that the largest component in magnitude (the first of the largest if there are ties) is made positive. Note that the eigenvectors are usually not the eigenvectors of the input matrix cov. They are the eigenvectors of the input matrix cov when the Principal Component method is used.

Returns — A double matrix containing the eigenvectors of the matrix from which the factors were extracted.

Imsl.Stat.RankException is thrown if the rank of the covariance matrix is less than the number of factors.

Imsl.Stat.NoDegreesOfFreedomException is thrown if there are no degrees of freedom for the significance testing.

Imsl.Stat.NotSemiDefiniteException is thrown if the Hessian matrix not semi-definite.

Imsl.Stat.NotPositiveSemiDefiniteException is thrown if the covariance matrix is not positive semi-definite.

Imsl.Stat.NotPositiveDefiniteException is thrown if the covariance matrix is not positive definite because a variable is linearly related to other variables.

Imsl.Stat.SingularException is thrown if the covariance matrix is singular.

Imsl.Stat.BadVarianceException is thrown if the input variance is not in the allowed range.

Imsl.Stat.EigenvalueException is thrown if an error occurred in calculating the eigenvalues of the adjusted (inverse) covariance matrix. Check the input covariance matrix.
Imsl.Stat.NonPositiveEigenvalueException is thrown if in alpha factor analysis an eigenvalue is not positive. As all eigenvalues corresponding to the factors must be positive, either the number of factors must be reduced, or new initial estimates for the unique variances must be given.

SetVariances

```csharp
public void SetVariances(double[] uniq)
Sets the unique variances.
```

If this member function is not called, the elements of `uniq` are set to 0.0. If the iterative methods fail for the unique variances used, new initial estimates should be tried. These may be obtained by use of another factoring method (use the final estimates from the new method as initial estimates in the old method). Another alternative is to call member function `VarianceEstimationMethod` and set the input argument to 0. This will cause the initial unique variances to be estimated by the code.

`uniq` A `double` array of length `nvar` containing the unique variances.

**Description**

Class `FactorAnalysis` computes principal components or initial factor loading estimates for a variance-covariance or correlation matrix using exploratory factor analysis models.

Models available are the principal component model for factor analysis and the common factor model with additions to the common factor model in alpha factor analysis and image analysis. Methods of estimation include principal components, principal factor, image analysis, unweighted least squares, generalized least squares, and maximum likelihood.

For the principal component model there are methods to compute the characteristic roots, characteristic vectors, standard errors for the characteristic roots, and the correlations of the principal component scores with the original variables. Principal components obtained from correlation matrices are the same as principal components obtained from standardized (to unit variance) variables.

The principal component scores are the elements of the vector $y = \Gamma^T x$ where $\Gamma$ is the matrix whose columns are the characteristic vectors (eigenvectors) of the sample covariance (or correlation) matrix and $x$ is the vector of observed (or standardized) random variables. The variances of the principal component scores are the characteristic roots (eigenvalues) of the covariance (correlation) matrix.

Asymptotic variances for the characteristic roots were first obtained by Girshick (1939) and are given more recently by Kendall, Stuart, and Ord (1983, page 331). These variances are computed either for variance-covariance matrices or for correlation matrices.

The correlations of the principal components with the observed (or standardized) variables are the same as the unrotated factor loadings obtained for the principal components model for factor analysis when a correlation matrix is input.

In the factor analysis model used for factor extraction, the basic model is given as

$$\Sigma = \Lambda\Lambda^T + \Psi$$

where $\Sigma$ is the $p \times p$ population covariance matrix. $\Lambda$ is the $p \times k$ matrix of
factor loadings relating the factors $f$ to the observed variables $x$, and $\Psi$ is the $p \times p$ matrix of covariances of the unique errors $e$. Here, $p$ represents the number of variables and $k$ is the number of factors. The relationship between the factors, the unique errors, and the observed variables is given as $x = \Lambda f + e$ where, in addition, it is assumed that the expected values of $e$, $f$, and $x$ are zero. (The sample means can be subtracted from $x$ if the expected value of $x$ is not zero.) It is also assumed that each factor has unit variance, the factors are independent of each other, and that the factors and the unique errors are mutually independent. In the common factor model, the elements of the vector of unique errors $e$ are also assumed to be independent of one another so that the matrix $\Psi$ is diagonal. This is not the case in the principal component model in which the errors may be correlated.

Further differences between the various methods concern the criterion that is optimized and the amount of computer effort required to obtain estimates. Generally speaking, the least-squares and maximum likelihood methods, which use iterative algorithms, require the most computer time with the principal factor, principal component, and the image methods requiring much less time since the algorithms in these methods are not iterative. The algorithm in alpha factor analysis is also iterative, but the estimates in this method generally require somewhat less computer effort than the least-squares and maximum likelihood estimates. In all algorithms one eigensystem analysis is required on each iteration.

**Example: Principal Components**

This example illustrates the use of the FactorAnalysis class for a nine-variable matrix. `FactorAnalysis.Model.PrincipalComponent` and input matrix type `FactorAnalysis.MatrixType.Correlation` are selected.

```csharp
using System;
using Imsl.Stat;
using PrintMatrix = Imsl.Math.PrintMatrix;
using PrintMatrixFormat = Imsl.Math.PrintMatrixFormat;
public class FactorAnalysisEx1
{
    public static void Main(String[] args)
    {
        double[,] corr = {
            { 1.0, 0.523, 0.395, 0.471,
              0.346, 0.426, 0.576, 0.434, 0.639},
            { 0.523, 1.0, 0.479, 0.506,
              0.418, 0.462, 0.547, 0.283, 0.645},
            { 0.395, 0.479, 1.0, 0.355,
              0.27, 0.254, 0.452, 0.219, 0.504},
            { 0.471, 0.506, 0.355, 1.0,
              0.691, 0.791, 0.443, 0.285, 0.505},
            { 0.346, 0.418, 0.27, 0.691,
              1.0, 0.679, 0.383, 0.149, 0.409},
            { 0.426, 0.462, 0.254, 0.791,
              0.679, 1.0, 0.372, 0.314, 0.472},
            { 0.576, 0.547, 0.452, 0.443,
              0.383, 0.372, 1.0, 0.385, 0.68},
            { 0.434, 0.283, 0.219, 0.285,
```

**Multivariate Analysis**

FactorAnalysis Class • 519
FactorAnalysis pc = new FactorAnalysis(corr,
   FactorAnalysis.MatrixType.Correlation, 9);
pc.FactorLoadingEstimationMethod =
   FactorAnalysis.Model.PrincipalComponent;
pc.DegreesOfFreedom = 100;

PrintMatrixFormat pmf = new PrintMatrixFormat();
   pmf.NumberFormat = "0.0000";
new PrintMatrix("Eigenvalues").Print(pmf, pc.GetValues());
new PrintMatrix("Percents").Print(pmf, pc.GetPercents());
new PrintMatrix
   ("Standard Errors").Print(pmf, pc.GetStandardErrors());
new PrintMatrix("Eigenvectors").Print(pmf, pc.GetVectors());
new PrintMatrix
   ("Unrotated Factor Loadings").Print(pmf, pc.GetFactorLoadings());

Output

Eigenvalues
   0
   0 4.6769
   1 1.2640
   2 0.8444
   3 0.5550
   4 0.4471
   5 0.4291
   6 0.3102
   7 0.2770
   8 0.1962

Percents
   0
   0 0.5197
   1 0.6601
   2 0.7539
   3 0.8156
   4 0.8653
   5 0.9130
   6 0.9474
   7 0.9782
   8 1.0000

Standard Errors
   0
   0 0.6498
   1 0.1771
   2 0.0986
Example: Factor Analysis

This example illustrates the use of the FactorAnalysis class. The following data were originally analyzed by Emmett (1949). There are 211 observations on 9 variables. Following Lawley and Maxwell (1971), three factors will be obtained by the method of maximum likelihood.

```csharp
using System;
using Imsl.Stat;
using PrintMatrix = Imsl.Math.PrintMatrix;
using PrintMatrixFormat = Imsl.Math.PrintMatrixFormat;

public class FactorAnalysisEx2
{
    public static void Main(String[] args)
    {
        double[,] cov = {
            {1.0, 0.523, 0.395, 0.471,
             0.346, 0.426, 0.576, 0.434, 0.639},
            {0.523, 1.0, 0.479, 0.506,
             0.418, 0.462, 0.547, 0.283, 0.645},
            {0.395, 0.479, 1.0, 0.365,
             0.8521, 0.1225, 0.0543};
```
```
FactorAnalysis fl = new FactorAnalysis(cov,  
    FactorAnalysis.MatrixType.VarianceCovariance, 3);  
fl.ConvergenceCriterion1 = .000001;  
fl.ConvergenceCriterion2 = .01;  
fl.FactorLoadingEstimationMethod =  
    FactorAnalysis.Model.MaximumLikelihood;  
fl.VarianceEstimationMethod = 0;  
fl.MaxStep = 10;  
fl.DegreesOfFreedom = 210;

PrintMatrixFormat pmf = new PrintMatrixFormat();  
pmf.NumberFormat = "0.0000";  
new PrintMatrix  
    ("Unique Error Variances").Print(pmf, fl.GetVariances());  
new PrintMatrix  
    ("Unrotated Factor Loadings").Print(pmf, fl.GetFactorLoadings());  
new PrintMatrix("Eigenvalues").Print(pmf, fl.GetValues());  
new PrintMatrix("Statistics").Print(pmf, fl.GetStatistics());
```
FactorAnalysis.MatrixType Enumeration

Matrix type.

public enumeration Imsl.Stat.FactorAnalysis.MatrixType

Fields

Correlation
  public Imsl.Stat.FactorAnalysis.MatrixType Correlation
  Indicates correlation matrix.

VarianceCovariance
  public Imsl.Stat.FactorAnalysis.MatrixType VarianceCovariance
  Indicates variance-covariance matrix.
FactorAnalysis.Model Enumeration

Model type.

public enumeration Imsl.Stat.FactorAnalysis.Model

Fields

AlphaFactorAnalysis
  Indicates alpha-factor analysis (common factor model) method used to obtain the estimates. Degrees of freedom is used for this estimation method.

GeneralizedLeastSquares
  Indicates generalized least-squares (common factor model) method used to obtain the estimates.

ImageFactorAnalysis
  Indicates Image-factor analysis (common factor model) method used to obtain the estimates.

MaximumLikelihood
  Indicates maximum likelihood method used to obtain the estimates. Degrees of freedom is used for this estimation method.

PrincipalComponent
  Indicates principal component (principal component model) used to obtain the estimates.

PrincipalFactor
  public Imsl.Stat.FactorAnalysis.Model PrincipalFactor
  Indicates principal factor (common factor model) will be used to obtain the estimates.

UnweightedLeastSquares
  Indicates unweighted least-squares (common factor model) method used to obtain the estimates. This option is the default.
DiscriminantAnalysis Class

Performs a linear or a quadratic discriminant function analysis among several known groups and the use of either reclassification, split sample, or the leaving-out-one methods in order to evaluate the rule.

public class Imsl.Stat.DiscriminantAnalysis

Properties

ClassificationMethod
public Imsl.Stat.DiscriminantAnalysis.Classification ClassificationMethod
{get; set; }
The classification method. Indicates the method of classification.
Use Classification member Reclassification or LeaveOutOne.
By default, Classification.Reclassification is used.

CovarianceComputation
{get; set; }
The type of covariance matrices to be computed. Indicates the type of covariance matrices to be computed.
Use CovarianceMatrix class member Pooled or PooledGroup.
By default, CovarianceMatrix.PooledGroup is used.

DiscriminationMethod
{get; set; }
The discrimination method. Indicates the method of discrimination.
Use Discrimination member Linear or Quadratic.
By default, Discrimination.Linear is used.

NRowsMissing
public int NRowsMissing {get; }
Returns the number of rows of data encountered containing missing values (NaN). An int representing the number of rows of data encountered containing missing values (NaN) for the classification, group, weight, and/or frequency variables.
If a row of data contains a missing value (NaN) for any of these variables, that row is excluded from the computations.

PriorType
public Imsl.Stat.DiscriminantAnalysis.PriorProbabilities PriorType {get; set; }
The type of prior probabilities to be computed. Indicates the type of prior probabilities to be computed.

Use `PriorProbabilities` member `PriorEqual` or `PriorProportional`.

By default, `PriorProbabilities.PriorEqual` is used.

**Constructor**

DiscriminantAnalysis

```csharp
public DiscriminantAnalysis(int nVariables, int nGroups)
```

Constructor for `DiscriminantAnalysis`.

- `nVariables` An `int` representing the number of variables to be used in the discrimination.
- `nGroups` An `int` representing the number of groups in the data.

**Methods**

GetClassMembership

```csharp
public int[] GetClassMembership()
```

Returns the group number to which the observation was classified.

If an observation has an invalid group number, frequency, or weight when the leaving-out-one method has been specified, then the observation is not classified and the corresponding elements of the array are set to zero.

Returns — An `int` array containing the group to which the observation was classified.

GetClassTable

```csharp
public double[,] GetClassTable()
```

Returns the classification table.

Each observation that is classified and has a group number equal to `1.0, 2.0, ..., nGroups` is entered into the table. The rows of the table correspond to the known group membership. The columns refer to the group to which the observation was classified.

Returns — A `nGroups × nGroups` `double` array containing the classification table.

GetCoefficients

```csharp
public double[,] GetCoefficients()
```

Returns the linear discriminant function coefficients.

The first column of the array contains the constant term, and the remaining columns contain the variable coefficients. The `i`-th row of the returned array corresponds to group `i`. The coefficients are always computed as linear discriminant function coefficients even when quadratic discrimination is specified.

Returns — A `double` array containing the linear discriminant function coefficients.
GetCovariance

public double[,] GetCovariance()
Returns the array of covariances.

Here, \( g = nGroups + 1 \) unless pooled only covariance matrices are computed, in which case \( g = 1 \). When pooled only covariance matrices are computed, the within-group covariance matrices are not computed. The pooled covariance matrix is always computed and is returned as the \( g \)-th covariance matrix.

Returns — A \( nVariables \times nVariables \times g \) double array containing the covariances.

GetGroupCounts

public int[] GetGroupCounts()
Returns the group counts.

Returns — An int array of length \( nGroups \) containing the number of observations in each group.

GetMahalanobis

public double[,] GetMahalanobis()
Returns the Mahalanobis distances between the group means.

For linear discrimination, the Mahalanobis distance

\[ D_{ij}^2 \]

between group means \( i \) and \( j \) is computed using the within covariance matrix for group \( i \) in place of the pooled covariance matrix.

Returns — A \( nGroups \times nGroups \) double array containing the Mahalanobis distances between the group means.

GetMeans

public double[,] GetMeans()
Returns the variable means.

The \( i \)-th row of the returned array contains the group \( i \) variable means.

Returns — A double array containing the variable means.

GetPrior

public double[] GetPrior()
Returns the prior probabilities for each group.

The elements of this vector should sum to 1.0. If this member function is not called, the elements are set so as to be equal if PriorType is set to PriorProbabilities.PriorEqual or they are set to be proportional to the sample size in each group if PriorType is set to PriorProbabilities.PriorProportional.

Returns — A double vector of length \( nGroups \) containing the prior probabilities for each group.

GetProbability

public double[,] GetProbability()
Returns the posterior probabilities for each observation.
Returns — A $x$.GetLength(0)×nGroups double array containing the posterior probabilities for each observation.

GetStatistics

```csharp
public double[] GetStatistics()
Returns statistics.
```

<table>
<thead>
<tr>
<th>$i$</th>
<th>Statistics[$i$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sum of the degrees of freedom for the within-covariance matrices.</td>
</tr>
<tr>
<td>1</td>
<td>Chi-squared statistic.</td>
</tr>
<tr>
<td>2</td>
<td>The degrees of freedom in the chi-squared statistic.</td>
</tr>
<tr>
<td>3</td>
<td>Probability of a greater chi-squared, respectively, of a test of the homogeneity of the within-covariance matrices. (Not computed when the pooled only covariance matrix is computed).</td>
</tr>
<tr>
<td>4 thru 4 + nGroups</td>
<td>Log of the determinant of each group's covariance matrix. (Not computed when the pooled only covariance matrix is computed) and of the pooled covariance matrix.</td>
</tr>
<tr>
<td>Last nGroups + 1 elements</td>
<td>Sum of the weights within each group.</td>
</tr>
<tr>
<td>Last element</td>
<td>Sum of the weights in all groups.</td>
</tr>
</tbody>
</table>

Returns — A double array containing output statistics.

SetPrior

```csharp
public void SetPrior(double[] prior)
Sets the prior probabilities for each group.
```

The elements of prior should sum to 1.0. If this member function is not called, the elements of prior are set so as to be equal if PriorType is set to PriorProbabilities.PriorEqual or they are set to be proportional to the sample size in each group if PriorType is set to PriorProbabilities.PriorProportional.

prior A double vector of length nGroups containing the prior probabilities for each group.

Update

```csharp
public void Update(double[,] x)
Processes a set of observations and performs a linear or quadratic discriminant function analysis among the several known groups.
```

The first nVariables columns correspond to the variables, and the last column (column nVariables) contains the group numbers. The groups must be numbered 1,2,..., nGroups.

x A double matrix containing the observations.
Imsl.Stat.SumOfWeightsNegException is thrown if the sum of the weights have become negative.

Imsl.Stat.EmptyGroupException is thrown if there are no observations in a group.

Cannot compute statistics.

Imsl.Stat.CovarianceSingularException is thrown if the variance-Covariance matrix is singular.

Imsl.Stat.PooledCovarianceSingularException is thrown if the pooled variance-Covariance matrix is singular.

Update

```java
public void Update(double[,] x, int groupIndex)
```
Processes a set of observations and performs a linear or quadratic discriminant function analysis among the several known groups.

The first nVariables columns correspond to the variables, excluding the groupIndex column. The groups must be numbered 1, 2, ..., nGroups.

x A double matrix containing the observations.

groupIndex An int containing the column index of x in which the group numbers are stored.

Multivariate Analysis DiscriminantAnalysis Class • 529

Imsl.Stat.SumOfWeightsNegException is thrown if the sum of the weights have become negative.

Imsl.Stat.EmptyGroupException is thrown if there are no observations in a group.

Cannot compute statistics.

Imsl.Stat.CovarianceSingularException is thrown if the variance-Covariance matrix is singular.

Imsl.Stat.PooledCovarianceSingularException is thrown if the pooled variance-Covariance matrix is singular.

Update

```java
public void Update(double[,] x, int[] varIndex)
```
Processes a set of observations and performs a linear or quadratic discriminant function analysis among the several known groups.

The columns indicated in varIndex correspond to the variables, and the last column (column nVariables) contains the group numbers. The groups must be numbered 1, 2, ..., nGroups.

x A double matrix containing the observations.

varIndex An int array containing the column indices in x that correspond to the variables to be used in the analysis.

Imsl.Stat.SumOfWeightsNegException is thrown if the sum of the weights have become negative.
Imsl.Stat.EmptyGroupException is thrown if there are no observations in a group.
Cannot compute statistics.

Imsl.Stat.CovarianceSingularException is thrown if the variance-Covariance matrix is singular.

Imsl.Stat.PooledCovarianceSingularException is thrown if the pooled variance-Covariance matrix is singular.

Update

```java
public void Update(double[,] x, double[] frequencies, double[] weights)
```
Processes a set of observations and associated frequencies and weights then performs a linear or quadratic discriminant function analysis among the several known groups.

The first nVariables columns correspond to the variables, and the last column (column nVariables) contains the group numbers. The groups must be numbered 1, 2, ..., nGroups.

x A double matrix containing the observations.

frequencies A double array containing the associated frequencies.

weights A double array containing the associated weights.

Imsl.Stat.SumOfWeightsNegException is thrown if the sum of the weights have become negative.

Imsl.Stat.EmptyGroupException is thrown if there are no observations in a group.
Cannot compute statistics.

Imsl.Stat.CovarianceSingularException is thrown if the variance-Covariance matrix is singular.

Imsl.Stat.PooledCovarianceSingularException is thrown if the pooled variance-Covariance matrix is singular.

Update

```java
public void Update(double[,] x, int groupIndex, int[] varIndex)
```
Processes a set of observations and performs a linear or quadratic discriminant function analysis among the several known groups.

The columns indicated in varIndex correspond to the variables, and groupIndex column contains the group numbers. The groups must be numbered 1, 2, ..., nGroups.

x A double matrix containing the observations.

groupIndex An int containing the column index of x in which the group numbers are stored.

varIndex An int array containing the column indices in x that correspond to the variables to be used in the analysis.

Imsl.Stat.SumOfWeightsNegException is thrown if the sum of the weights have become negative.
Imsl.Stat.EmptyGroupException is thrown if there are no observations in a group.
Cannot compute statistics.
Imsl.Stat.CovarianceSingularException is thrown if the variance-Covariance matrix
is singular.
Imsl.Stat.PooledCovarianceSingularException is thrown if the pooled variance-Covariance matrix is singular.

Update

public void Update(double[,] x, int groupIndex, double[] frequencies,
                    double[] weights)
Processes a set of observations and associated frequencies and weights then performs a linear or quadratic discriminant function analysis among the several known groups.
The first nVariables columns correspond to the variables, excluding the groupIndex column. The groups must be numbered 1,2, ..., nGroups.

x  A double matrix containing the observations.
groupIndex  An int containing the column index of x in which the group numbers are stored.
frequencies  A double array containing the associated frequencies.
weights  A double array containing the associated weights.

Imsl.Stat.SumOfWeightsNegException is thrown if the sum of the weights have become negative.
Imsl.Stat.EmptyGroupException is thrown if there are no observations in a group.
Cannot compute statistics.
Imsl.Stat.CovarianceSingularException is thrown if the variance-Covariance matrix
is singular.
Imsl.Stat.PooledCovarianceSingularException is thrown if the pooled variance-Covariance matrix is singular.

Update

public void Update(double[,] x, int[] varIndex, double[] frequencies,
                    double[] weights)
Processes a set of observations and associated frequencies and weights then performs a linear or quadratic discriminant function analysis among the several known groups.
The columns indicated in varIndex correspond to the variables, and the last column (column nVariables) contains the group numbers. The groups must be numbered 1,2, ..., nGroups.

x  A double matrix containing the observations.
varIndex  An int array containing the column indices in x that correspond to the variables to be used in the analysis.
frequencies  A double array containing the associated frequencies.
weights  A double array containing the associated weights.

Imsl.Stat.SumOfWeightsNegException is thrown if the sum of the weights have become negative.

Imsl.Stat.EmptyGroupException is thrown if there are no observations in a group.

Cannot compute statistics.

Imsl.Stat.CovarianceSingularException is thrown if the variance-Covariance matrix is singular.

Imsl.Stat.PooledCovarianceSingularException is thrown if the pooled variance-Covariance matrix is singular.

Update

public void Update(double[,] x, int groupIndex, int[] varIndex, double[] frequencies, double[] weights)

Processes a set of observations and associated frequencies and weights then performs a linear or quadratic discriminant function analysis among the several known groups.

The columns indicated in varIndex correspond to the variables, and groupIndex column contains the group numbers. The groups must be numbered 1,2,..., nGroups.

x  A double matrix containing the observations.

groupIndex  An int containing the column index of x in which the group numbers are stored.

varIndex  An int array containing the column indices in x that correspond to the variables to be used in the analysis.

frequencies  A double array containing the associated frequencies.

weights  A double array containing the associated weights.

Imsl.Stat.SumOfWeightsNegException is thrown if the sum of the weights have become negative.

Imsl.Stat.EmptyGroupException is thrown if there are no observations in a group.

Cannot compute statistics.

Imsl.Stat.CovarianceSingularException is thrown if the variance-Covariance matrix is singular.

Imsl.Stat.PooledCovarianceSingularException is thrown if the pooled variance-Covariance matrix is singular.

Description

Class DiscriminantAnalysis performs discriminant function analysis using either linear or quadratic discrimination. The output from DiscriminantAnalysis includes a measure of distance between the groups, a table summarizing the classification results, a matrix containing the posterior probabilities of group membership for each observation, and the within-sample
means and covariance matrices. The linear discriminant function coefficients are also computed.

All observations are input during one call to DiscriminantAnalysis, a method of operation that has the advantage of simplicity.

All observations in \( x \) are used to compute the means. The covariance matrices are factored. Requested statistics of interest are computed: the linear discriminant functions, the prior probabilities, the log of the determinant of each of the covariance matrices, a test statistic for testing that all of the within-group covariance matrices are equal, and a matrix of Mahalanobis distances between the groups. The matrix of Mahalanobis distances is computed via the pooled covariance matrix when linear discrimination is specified, the row covariance matrix is used when the discrimination is quadratic. Covariance matrices are defined as follows. Let \( N_i \) denote the sum of the frequencies of the observations in group \( i \), and let \( M_i \) denote the number of observations in group \( i \). Then, if \( S_i \) denotes the within-group \( i \) covariance matrix,

\[
S_i = \frac{1}{N_i - 1} \sum_{j=1}^{M_i} w_j f_j (x_j - \overline{x})(x_j - \overline{x})^T
\]

where \( w_j \) is the weight of the \( j \)-th observation in group \( i \), \( f_j \) is its frequency, \( x_j \) is the \( j \)-th observation column vector (in group \( i \)), and \( \overline{x} \) denotes the mean vector of the observations in group \( i \). The mean vectors are computed as

\[
\overline{x} = \frac{1}{W_i} \sum_{j=1}^{M_i} w_j f_j x_j
\]

where

\[
W_i = \sum_{j=1}^{M_i} w_j f_j
\]

Given the means and the covariance matrices, the linear discriminant function for group \( i \) is computed as:

\[
z_i = \ln(p_i) - 0.5\overline{x}_i^T S_p^{-1} \overline{x}_i + x^T S_p^{-1} \overline{x}_i
\]

where \( \ln(p_i) \) is the natural log of the prior probability for the \( i \)-th group, \( x \) is the observation to be classified, and \( S_p \) denotes the pooled covariance matrix.

Let \( S \) denote either the pooled covariance matrix or one of the within-group covariance matrices \( S_i \). (\( S \) will be the pooled covariance matrix in linear discrimination, and \( S_i \) otherwise.) The Mahalanobis distance between group \( i \) and group \( j \) is computed as:

\[
D_{ij}^2 = (\overline{x}_i - \overline{x}_j)^T S^{-1}(\overline{x}_i - \overline{x}_j)
\]

Finally, the asymptotic chi-squared test for the equality of covariance matrices is computed as follows (Morrison 1976, page 252):

\[
\gamma = C^{-1} \sum_{i=1}^{k} n_i \{ \ln(|S_p|) - \ln(|S_i|) \}
\]
where \( n_i \) is the number of degrees of freedom in the \( i \)-th sample covariance matrix, \( k \) is the number of groups, and

\[
C^{-1} = 1 - \frac{2p^2 + 3p - 1}{6(p+1)(k-1)} \left( \sum_{i=1}^{k} \frac{1}{n_i} - \frac{1}{\sum_j n_j} \right)
\]

where \( p \) is the number of variables.

The estimated posterior probability of each observation \( x \) belonging to group \( i \) is computed using the prior probabilities and the sample mean vectors and estimated covariance matrices under a multivariate normal assumption. Under quadratic discrimination, the within-group covariance matrices are used to compute the estimated posterior probabilities. The estimated posterior probability of an observation \( x \) belonging to group \( i \) is

\[
\hat{q}_i(x) = \frac{e^{-\frac{1}{2}D_i^2(x)}}{\sum_{j=1}^{k} e^{-\frac{1}{2}D_j^2(x)}}
\]

where

\[
D_i^2(x) = \begin{cases} 
(x - \bar{x}_i)^T S_i^{-1} (x - \bar{x}_i) + \ln |S_i| - 2\ln(p_i) & \text{LINEAR or QUADRATIC} \\
(x - \bar{x}_i)^T S_p^{-1} (x - \bar{x}_i) - 2\ln(p_i) & \text{LINEAR POOLED}
\end{cases}
\]

For the leaving-out-one method of classification, the sample mean vector and sample covariance matrices in the formula for \( D_i^2(x) \) are adjusted so as to remove the observation \( x \) from their computation. For linear discrimination, the linear discriminant function coefficients are actually used to compute the same posterior probabilities.

Using the posterior probabilities, each observation in \( X \) is classified into a group; the result is tabulated in the matrix \( \text{CLASS} \) and saved in the vector \( \text{ICLASS} \). \( \text{CLASS} \) is not altered at this stage if \( X(\cdot, \text{IGRP}) \) contains a group number that is out of range. If the reclassification method is specified, then all observations with no missing values in the \text{nVariables} classification variables are classified. When the leaving-out-one method is used, observations with invalid group numbers, weights, frequencies or classification variables are not classified. Regardless of the frequency, a 1 is added (or subtracted) from \( \text{CLASS} \) for each row of \( X \) that is classified and contains a valid group number. When the leaving-out-one method is used, adjustment is made to the posterior probabilities to remove the effect of the observation in the classification rule. In this adjustment, each observation is presumed to have a weight of \( X(\cdot, \text{IWT}) \), if \( \text{IWT} > 0 \) and a frequency of 1.0. See Lachenbruch (1975, page 36) for the required adjustment.

Finally, upon completion, the covariance matrices are computed from their LU factorizations.

**Example: Discriminant Analysis**

This example uses linear discrimination with equal prior probabilities on Fisher’s (1936) iris data. This example illustrates the use of the DiscriminantAnalysis class.
using System;
using Imsl.Stat;
using PrintMatrix = Imsl.Math.PrintMatrix;

public class DiscriminantAnalysisEx1
{
    public static void Main(String[] args)
    {
        double[,] xorig = {
            {1.0, 5.1, 3.5, 1.4, .2},
            {1.0, 4.9, 3.0, 1.4, .2},
            {1.0, 4.7, 3.2, 1.3, .2},
            {1.0, 4.6, 3.1, 1.5, .2},
            {1.0, 5.0, 3.6, 1.4, .2},
            {1.0, 5.4, 3.9, 1.7, .4},
            {1.0, 4.6, 3.4, 1.4, .3},
            {1.0, 5.0, 3.4, 1.5, .2},
            {1.0, 4.4, 2.9, 1.4, .2},
            {1.0, 4.9, 3.1, 1.5, .1},
            {1.0, 5.4, 3.7, 1.5, .2},
            {1.0, 4.8, 3.4, 1.6, .2},
            {1.0, 4.8, 3.0, 1.4, .1},
            {1.0, 4.3, 3.0, 1.1, .1},
            {1.0, 5.8, 4.0, 1.2, .2},
            {1.0, 5.7, 4.4, 1.5, .4},
            {1.0, 5.4, 3.9, 1.3, .4},
            {1.0, 5.1, 3.5, 1.4, .3},
            {1.0, 5.7, 3.8, 1.7, .3},
            {1.0, 5.1, 3.8, 1.5, .3},
            {1.0, 5.4, 3.4, 1.7, .2},
            {1.0, 5.1, 3.7, 1.5, .4},
            {1.0, 4.6, 3.6, 1.0, .2},
            {1.0, 5.1, 3.3, 1.7, .5},
            {1.0, 4.8, 3.4, 1.9, .2},
            {1.0, 5.0, 3.0, 1.6, .2},
            {1.0, 5.0, 3.4, 1.6, .4},
            {1.0, 5.2, 3.5, 1.5, .2},
            {1.0, 5.2, 3.4, 1.4, .2},
            {1.0, 4.7, 3.2, 1.6, .2},
            {1.0, 4.8, 3.1, 1.6, .2},
            {1.0, 5.4, 3.4, 1.5, .4},
            {1.0, 5.2, 4.1, 1.5, .1},
            {1.0, 5.5, 4.2, 1.4, .2},
            {1.0, 4.9, 3.1, 1.5, .2},
            {1.0, 5.0, 3.2, 1.2, .2},
            {1.0, 5.5, 3.5, 1.3, .2},
            {1.0, 4.9, 3.6, 1.4, .1},
            {1.0, 4.4, 3.0, 1.3, .2},
            {1.0, 5.1, 3.4, 1.5, .2},
            {1.0, 5.0, 3.5, 1.3, .3},
            {1.0, 4.5, 2.3, 1.3, .3},
            {1.0, 4.4, 3.2, 1.3, .2},
            {1.0, 5.0, 3.5, 1.6, .6},
            {1.0, 5.1, 3.8, 1.9, .4},
            {1.0, 4.8, 3.0, 1.4, .3},
            {1.0, 5.1, 3.8, 1.6, .2},
        };
    }
}
{1.0, 4.6, 3.2, 1.4, .2},
{1.0, 5.3, 3.7, 1.5, .2},
{1.0, 5.0, 3.3, 1.4, .2},
{2.0, 7.0, 3.2, 4.7, 1.4},
{2.0, 6.4, 3.2, 4.6, 1.5},
{2.0, 6.9, 3.1, 4.9, 1.5},
{2.0, 5.5, 2.3, 4.0, 1.3},
{2.0, 6.5, 2.8, 4.6, 1.5},
{2.0, 5.7, 2.8, 4.5, 1.3},
{2.0, 6.3, 3.3, 4.7, 1.6},
{2.0, 4.9, 2.4, 3.3, 1.0},
{2.0, 6.6, 2.9, 4.6, 1.3},
{2.0, 5.2, 2.7, 3.9, 1.4},
{2.0, 5.0, 2.0, 3.5, 1.0},
{2.0, 5.9, 3.0, 4.2, 1.5},
{2.0, 6.0, 2.2, 4.0, 1.0},
{2.0, 6.1, 2.9, 4.7, 1.4},
{2.0, 5.6, 2.9, 3.6, 1.3},
{2.0, 6.7, 3.1, 4.4, 1.4},
{2.0, 5.6, 3.0, 4.5, 1.5},
{2.0, 5.8, 2.7, 4.1, 1.0},
{2.0, 6.2, 2.2, 4.5, 1.5},
{2.0, 5.6, 2.5, 3.9, 1.1},
{2.0, 5.9, 3.2, 4.8, 1.8},
{2.0, 6.1, 2.8, 4.0, 1.3},
{2.0, 6.3, 2.5, 4.9, 1.5},
{2.0, 6.1, 2.8, 4.7, 1.2},
{2.0, 6.4, 2.9, 4.3, 1.3},
{2.0, 6.6, 3.0, 4.4, 1.4},
{2.0, 6.8, 2.8, 4.8, 1.4},
{2.0, 6.7, 3.0, 5.0, 1.7},
{2.0, 6.0, 2.9, 4.5, 1.5},
{2.0, 5.7, 2.6, 3.5, 1.0},
{2.0, 5.5, 2.4, 3.8, 1.1},
{2.0, 5.5, 2.4, 3.7, 1.0},
{2.0, 5.8, 2.7, 3.9, 1.2},
{2.0, 6.0, 2.7, 5.1, 1.6},
{2.0, 5.4, 3.0, 4.5, 1.5},
{2.0, 6.0, 3.4, 4.5, 1.6},
{2.0, 6.7, 3.1, 4.7, 1.5},
{2.0, 6.3, 2.3, 4.4, 1.3},
{2.0, 5.6, 3.0, 4.1, 1.3},
{2.0, 5.5, 2.5, 4.0, 1.3},
{2.0, 5.5, 2.6, 4.4, 1.2},
{2.0, 6.1, 3.0, 4.6, 1.4},
{2.0, 5.8, 2.6, 4.0, 1.2},
{2.0, 5.0, 2.3, 3.3, 1.0},
{2.0, 5.6, 2.7, 4.2, 1.3},
{2.0, 5.7, 3.0, 4.2, 1.2},
{2.0, 5.7, 2.9, 4.2, 1.3},
{2.0, 6.2, 2.9, 4.3, 1.3},
{2.0, 5.1, 2.5, 3.0, 1.1},
{2.0, 5.7, 2.8, 4.1, 1.3},
{3.0, 6.3, 3.3, 6.0, 2.5},
{3.0, 5.8, 2.7, 5.1, 1.9},
{3.0, 7.1, 3.0, 5.9, 2.1},

536 • DiscriminantAnalysis Class IMSL C# Numerical Library
int[] ipermu = new int[]{2, 3, 4, 5, 1};

double[,] x = new double[xorig.GetLength(0), xorig.GetLength(1)];

for (int i = 0; i < xorig.GetLength(0); i++)
{
    for (int j = 1; j < xorig.GetLength(1); j++)
    {
        Multivariate Analysis DiscriminantAnalysis Class • 537
\{
    x[i,j - 1] = xorig[i,j];
\}

for (int i = 0; i < xorig.GetLength(0); i++)
\{ 
    x[i,4] = xorig[i,0];
\}

int nvar = x.GetLength(1) - 1;

DiscriminantAnalysis da = new DiscriminantAnalysis(nvar, 3);
da.Update(x);

new PrintMatrix("Xmean are: ").SetPageWidth(80).Print(da.GetMeans());
new PrintMatrix("Coef: ").SetPageWidth(80).Print(da.GetCoefficients());
new PrintMatrix("ClassMembership: ").SetPageWidth(80).Print(da.GetClassMembership());
new PrintMatrix("ClassTable: ").SetPageWidth(80).Print(da.GetClassTable());
double[,] cov = da.GetCovariance();
double[,] tmpCov = new double[cov.GetLength(1), cov.GetLength(2)];
for (int i = 0; i < cov.GetLength(0); i++)
\{
    for (int j = 0; j < cov.GetLength(1); j++)
        for (int k = 0; k < cov.GetLength(2); k++)
            tmpCov[j, k] = cov[i, j, k];
    new PrintMatrix("Covariance Matrix " + i + " : ").SetPageWidth(80).Print(tmpCov);
\}
new PrintMatrix("MAHALANOBIS: ").SetPageWidth(80).Print(da.GetMahalanobis());
Console.Out.WriteLine("nrmiss = " + da.NRowsMissing);
\}
\}

Output

\begin{verbatim}
Xmean are:
0 5.006 3.428 1.462 0.246
1 5.936 2.77 4.26 1.326
2 6.588 2.974 5.552 2.026

Coef:
0 -86.308469973674 23.5441667229203 23.5878704955898 -16.4306390229439
1 -72.8526074006422 15.6982090760379 7.02509844329562 5.21145093416415
\end{verbatim

538 • DiscriminantAnalysis Class IML C# Numerical Library
2  -104.36831998645  12.4458489937766  3.68527961207532  12.7665449735348

4
0  -17.3984107815644
1   6.43422920040657
2   21.0791130134185

Counts:
  0  50
  1  50
  2  50

Stats:
  0
  0  147
  1  NaN
  2  NaN
  3  NaN
  4  NaN
  5  NaN
  6  NaN
  7  -9.95853877004797
  8  50
  9  50
 10  50
 11  150

ClassMembership:
  0
  0  1
  1  1
  2  1
  3  1
  4  1
  5  1
  6  1
  7  1
  8  1
  9  1
 10  1
 11  1
 12  1
 13  1
 14  1
 15  1
 16  1
 17  1
 18  1
 19  1
 20  1
 21  1
 22  1
 23  1
 24  1
 25  1

Multivariate Analysis DiscriminantAnalysis Class • 539
DiscriminantAnalysis Class

IMSL C# Numerical Library
Multivariate Analysis

DiscriminantAnalysis Class • 541
ClassTable:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>48</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>49</td>
</tr>
</tbody>
</table>

Covariance Matrix 0:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.265008163265306</td>
<td>0.0927210884353742</td>
<td>0.167514285714286</td>
</tr>
<tr>
<td>1</td>
<td>0.0927210884353742</td>
<td>0.115387755102041</td>
<td>0.055243537414966</td>
</tr>
<tr>
<td>2</td>
<td>0.167514285714286</td>
<td>0.055243537414966</td>
<td>0.185187755102041</td>
</tr>
<tr>
<td>3</td>
<td>0.0384013605442177</td>
<td>0.0327102040816327</td>
<td>0.042665306122449</td>
</tr>
<tr>
<td>4</td>
<td>0.0384013605442177</td>
<td>0.0327102040816327</td>
<td>0.042665306122449</td>
</tr>
<tr>
<td>5</td>
<td>0.042665306122449</td>
<td>0.0418816326530612</td>
<td>0.0418816326530612</td>
</tr>
<tr>
<td>6</td>
<td>0.333333333333333</td>
<td>0.333333333333333</td>
<td>0.333333333333333</td>
</tr>
<tr>
<td>7</td>
<td>0.333333333333333</td>
<td>0.333333333333333</td>
<td>0.333333333333333</td>
</tr>
<tr>
<td>8</td>
<td>0.999999999999999</td>
<td>0.9928356178352E-34</td>
<td>0.9928356178352E-34</td>
</tr>
<tr>
<td>9</td>
<td>1.11180260918759E-18</td>
<td>2.72405964325484E-38</td>
<td>2.72405964325484E-38</td>
</tr>
<tr>
<td>10</td>
<td>1.8527748898975E-23</td>
<td>3.23708368191298E-44</td>
<td>3.23708368191298E-44</td>
</tr>
<tr>
<td>11</td>
<td>1.62164851137697E-18</td>
<td>1.83320074383666E-37</td>
<td>1.83320074383666E-37</td>
</tr>
<tr>
<td>12</td>
<td>1.45922504711622E-18</td>
<td>3.2625064352377E-38</td>
<td>3.2625064352377E-38</td>
</tr>
<tr>
<td>13</td>
<td>1.11721885779029E-19</td>
<td>1.31664193135497E-39</td>
<td>1.31664193135497E-39</td>
</tr>
<tr>
<td>14</td>
<td>5.4873987251784E-30</td>
<td>1.5312647295902E-52</td>
<td>1.5312647295902E-52</td>
</tr>
<tr>
<td>15</td>
<td>1.26150509583788E-27</td>
<td>2.268704278047E-48</td>
<td>2.268704278047E-48</td>
</tr>
</tbody>
</table>

Prior:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.333333333333333</td>
<td>0.333333333333333</td>
</tr>
<tr>
<td>1</td>
<td>0.333333333333333</td>
<td>0.333333333333333</td>
</tr>
<tr>
<td>2</td>
<td>0.333333333333333</td>
<td>0.333333333333333</td>
</tr>
</tbody>
</table>

PROB:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.89635792768677E-22</td>
<td>2.6116827494833E-42</td>
<td>2.6116827494833E-42</td>
</tr>
<tr>
<td>1</td>
<td>7.2179691863879E-18</td>
<td>5.04214334588401E-37</td>
<td>5.04214334588401E-37</td>
</tr>
<tr>
<td>2</td>
<td>1.4638494952907E-19</td>
<td>4.67593159333071E-39</td>
<td>4.67593159333071E-39</td>
</tr>
<tr>
<td>3</td>
<td>1.26853637674403E-16</td>
<td>3.56661049202016E-35</td>
<td>3.56661049202016E-35</td>
</tr>
<tr>
<td>4</td>
<td>1.6373874612726E-22</td>
<td>1.08260526717561E-42</td>
<td>1.08260526717561E-42</td>
</tr>
<tr>
<td>5</td>
<td>3.88328166174543E-21</td>
<td>4.56654013405467E-40</td>
<td>4.56654013405467E-40</td>
</tr>
<tr>
<td>6</td>
<td>1.11346944585999E-18</td>
<td>2.3026848348845E-37</td>
<td>2.3026848348845E-37</td>
</tr>
<tr>
<td>7</td>
<td>3.87758657727045E-20</td>
<td>1.07449600387617E-39</td>
<td>1.07449600387617E-39</td>
</tr>
<tr>
<td>8</td>
<td>0.999999999999999</td>
<td>9.4829356178352E-34</td>
<td>9.4829356178352E-34</td>
</tr>
<tr>
<td>9</td>
<td>1.11180260918759E-18</td>
<td>2.72405964325484E-38</td>
<td>2.72405964325484E-38</td>
</tr>
<tr>
<td>10</td>
<td>1.8527748898975E-23</td>
<td>3.23708368191298E-44</td>
<td>3.23708368191298E-44</td>
</tr>
<tr>
<td>11</td>
<td>1.62164851137697E-18</td>
<td>1.83320074383666E-37</td>
<td>1.83320074383666E-37</td>
</tr>
<tr>
<td>12</td>
<td>1.45922504711622E-18</td>
<td>3.2625064352377E-38</td>
<td>3.2625064352377E-38</td>
</tr>
<tr>
<td>13</td>
<td>1.11721885779029E-19</td>
<td>1.31664193135497E-39</td>
<td>1.31664193135497E-39</td>
</tr>
<tr>
<td>14</td>
<td>5.4873987251784E-30</td>
<td>1.5312647295902E-52</td>
<td>1.5312647295902E-52</td>
</tr>
<tr>
<td>15</td>
<td>1.26150509583788E-27</td>
<td>2.268704278047E-48</td>
<td>2.268704278047E-48</td>
</tr>
<tr>
<td></td>
<td>1.59433806215665E-25</td>
<td>3.86827125844699E-45</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---------------------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>4.42374070046694E-21</td>
<td>1.2243307558763E-40</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>1.77431503515482E-22</td>
<td>2.5521523423363E-42</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>2.52323792183582E-22</td>
<td>5.79207872447459E-42</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1.27463856628519E-19</td>
<td>4.36777421186783E-39</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>1.46599007679990E-20</td>
<td>1.98724136674322E-39</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>6.56928044945199E-25</td>
<td>7.76917736630943E-46</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>0.9999999999999999</td>
<td>8.91234785423208E-15</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>0.9999999999999999</td>
<td>1.07070246199648E-16</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>1.97339035989265E-16</td>
<td>5.71026880713927E-35</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>3.97731835976817E-17</td>
<td>4.37623934002496E-45</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>1.54816504878351E-21</td>
<td>1.59535976676668E-41</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>9.27184652389738E-22</td>
<td>6.29795546615504E-42</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>6.65144223645628E-22</td>
<td>2.97794711204608E-35</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>2.29993857694263E-16</td>
<td>7.18266552067294E-35</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>1.97540361007101E-19</td>
<td>2.78833402097428E-38</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>7.10004087420799E-27</td>
<td>2.16408315802475E-48</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>1.61029436549486E-28</td>
<td>2.74373397405663E-50</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>1.2052193403381E-17</td>
<td>1.27724507978322E-36</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>1.59718567904273E-21</td>
<td>9.03377178193152E-42</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>1.93986880928699E-24</td>
<td>1.66280764122995E-45</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>3.31023376400147E-23</td>
<td>7.00097072116312E-44</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>4.19024193543482E-17</td>
<td>6.99144060854016E-46</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>1.76935863474539E-20</td>
<td>3.54169362585279E-40</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>1.06301365121764E-21</td>
<td>2.00386616263149E-41</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>2.17421702175081E-11</td>
<td>2.13780794563050E-28</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>1.54075327236441E-18</td>
<td>1.30518608332622E-37</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>8.94058875293973E-16</td>
<td>1.31551062253732E-32</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>1.61620262204115E-19</td>
<td>3.20599205920815E-35</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>7.17443172160176E-16</td>
<td>7.17243513472585E-35</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>2.083089328107E-22</td>
<td>2.29978349710803E-42</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>2.794821528124E-18</td>
<td>2.62953861269042E-37</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>5.97560355678576E-23</td>
<td>9.82081977684015E-44</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>2.32225794022777E-20</td>
<td>4.24175670110456E-40</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>1.96973175506613E-18</td>
<td>0.99988412240982</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>1.2428779621613E-19</td>
<td>0.99925740339892</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>2.08826305422312E-22</td>
<td>0.99580694715406</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>2.1988894390163E-22</td>
<td>0.99964239804226</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>4.21367813030423E-23</td>
<td>0.99559034514412</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>8.12728652949083E-23</td>
<td>0.99850201836847</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>3.54989967813166E-22</td>
<td>0.98583457963257</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>5.00706454655384E-14</td>
<td>0.99999888018824</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>5.8333339389098E-20</td>
<td>0.99987130527527</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>1.24103873949892E-20</td>
<td>0.99950269181115</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>1.95682763937994E-18</td>
<td>0.9999987915944</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>5.46890036424978E-20</td>
<td>0.99922942836129</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>2.71612817142455E-18</td>
<td>0.99998779830561</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>1.24444523187668E-23</td>
<td>0.99342574139504</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>5.57431270513181E-14</td>
<td>1.64921547359976</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>2.36961149493997E-17</td>
<td>4.26821220373998</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>8.4293281034778E-24</td>
<td>0.98064710843801</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>2.50570161487087E-16</td>
<td>0.9999908482366</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>1.67035204315192E-27</td>
<td>0.95957347246689</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>1.34150265115522E-17</td>
<td>0.99999670389456</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>7.40811758162493E-28</td>
<td>0.25322822473817</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>9.39929180867634E-17</td>
<td>0.99990654708731</td>
<td></td>
</tr>
</tbody>
</table>

**Multivariate Analysis DiscriminantAnalysis Class • 543**
MAHALANOBIS:

\[
\begin{array}{ccc}
0 & 1 & 2 \\
0 & 89.8641855820738 & 179.384712514278 \\
1 & 89.8641855820738 & 0 \\
2 & 179.384712514278 & 17.20106428396 \\
\end{array}
\]

nrmiss = 0

---

**DiscriminantAnalysis.Discrimination Enumeration**

**Discrimination Methods.**

**Fields**

- Linear
  ```
  public Imsl.Stat.DiscriminantAnalysis.Discrimination Linear 
  Indicates a linear discrimination method.
  ```

- Quadratic
  ```
  Indicates a quadratic discrimination method.
  ```
**DiscriminantAnalysis.CovarianceMatrix Enumeration**

Covariance Matrix type.

```csharp
```

**Fields**

- **Pooled**
  ```csharp
  Indicates Pooled covariances computed.
  ```

- **PooledGroup**
  ```csharp
  Indicates Pooled, group covariances computed.
  ```

**DiscriminantAnalysis.Classification Enumeration**

Classification Method.

```csharp
public enumeration Imsl.Stat.DiscriminantAnalysis.Classification
```

**Fields**

- **LeaveOutOne**
  ```csharp
  public Imsl.Stat.DiscriminantAnalysis.Classification LeaveOutOne
  Indicates leave-out-one as the classification method.
  ```

- **Reclassification**
  ```csharp
  public Imsl.Stat.DiscriminantAnalysis.Classification Reclassification
  Indicates reclassification as the classification method.
  ```
DiscriminantAnalysis.PriorProbabilities Enumeration

Prior probabilities type.


Fields

PriorEqual
Indicates prior probability type is to be prior equal.

PriorProportional
Indicates prior probability type is to be prior proportional.
Chapter 20: Probability
Distribution Functions
and Inverses

Types

class Cdf ................................................................. 551
interface ICdfFunction ............................................. 572
class InverseCdf ....................................................... 572

Usage Notes

Definitions and discussions of the terms basic to this chapter can be found in Johnson and Kotz (1969, 1970a, 1970b). These are also good references for the specific distributions.

In order to keep the calling sequences simple, whenever possible, the methods/classes described in this chapter are written for standard forms of statistical distributions. Hence, the number of parameters for any given distribution may be fewer than the number often associated with the distribution. Also, the methods relating to the normal distribution, Cdf.Normal and Cdf.InverseNormal, are for a normal distribution with mean equal to zero and variance equal to one. For other means and variances, it is very easy for the user to standardize the variables by subtracting the mean and dividing by the square root of the variance.

The distribution function for the (real, single-valued) random variable $X$ is the function $F$ defined for all real $x$ by

$$ F(x) = \text{Prob}(X \leq x) $$

where Prob($\cdot$) denotes the probability of an event. The distribution function is often called the cumulative distribution function (CDF).

For distributions with finite ranges, such as the beta distribution, the CDF is 0 for values less than the left endpoint and 1 for values greater than the right endpoint. The methods in the Cdf classes described in this chapter return the correct values for the distribution functions.
when values outside of the range of the random variable are input, but warning error conditions are set in these cases.

**Discrete Random Variables**

For discrete distributions, the function giving the probability that the random variable takes on specific values is called the *probability function*, defined by

\[ p(x) = \text{Prob}(X = x) \]

The CDF for a discrete random variable is

\[ F(x) = \sum_{A} p(k) \]

where \( A \) is set such that \( k \leq x \). Since the distribution function is a step function, its inverse does not exist uniquely.

**Continuous Distributions**

For continuous distributions, a probability function, as defined above, would not be useful because the probability of any given point is 0. For such distributions, the useful analog is the *probability density function* (PDF). The integral of the PDF is the probability over the interval, if the continuous random variable \( X \) has PDF \( f \), then

\[ \text{Prob}(a \leq X \leq b) = \int_{a}^{b} f(x) \, dx \]

The relationship between the CDF and the PDF is

\[ F(x) = \int_{-\infty}^{x} f(t) \, dt \]

For (absolutely) continuous distributions, the value of \( F(x) \) uniquely determines \( x \) within the support of the distribution. The "Inverse" methods in the \texttt{Cdf} class compute the inverses of the distribution functions. That is, given \( F(x) \) (called "prob" for "probability"), a method such as, \texttt{InverseBeta} in the \texttt{Cdf} class computes \( x \). The inverses are defined only over the open interval \((0,1)\).

**Additional Comments**

Whenever a probability close to 1.0 results from a call to a distribution function or is to be input to an inverse function, it is often impossible to achieve good accuracy because of the
nature of the representation of numeric values. In this case, it may be better to work with the complementary distribution function (one minus the distribution function). If the distribution is symmetric about some point (as the normal distribution, for example) or is reflective about some point (as the beta distribution, for example), the complementary distribution function has a simple relationship with the distribution function. For example, to evaluate the standard normal distribution at 4.0, using the `Normal` method in the `Cdf` class directly, the result to six places is 0.999968. Only two of those digits are really useful, however. A more useful result may be 1.000000 minus this value, which can be obtained to six places as 3.16712e-05 by evaluating `Normal` at -4.0. For the normal distribution, the two values are related by \( \Phi(x) = 1 - \Phi(-x) \), where \( \Phi(\cdot) \) is the normal distribution function. Another example is the beta distribution with parameters 2 and 10. This distribution is skewed to the right, so evaluating `Beta` at 0.7, 0.999953 is obtained. A more precise result is obtained by evaluating `Beta` with parameters 10 and 2 at 0.3. This yields 4.72392e-5.

Many of the algorithms used by the classes in this chapter are discussed by Abramowitz and Stegun (1964). The algorithms make use of various expansions and recursive relationships and often use different methods in different regions.

Cumulative distribution functions are defined for all real arguments. However, if the input to one of the distribution functions in this chapter is outside the range of the random variable, an error is issued.

---

### Cdf Class

Cumulative distribution functions.

```java
public class Imsl.Stat.Cdf
```

#### Methods

**Beta**

```java
static public double Beta(double x, double pin, double qin)
```

Evaluates the beta probability distribution function.

Method `Beta` evaluates the distribution function of a beta random variable with parameters `pin` and `qin`. This function is sometimes called the *incomplete beta ratio* and, with \( p = pin \) and \( q = qin \), is denoted by \( I_x(p, q) \). It is given by

\[
I_x(p, q) = \frac{\Gamma(p) \Gamma(q)}{\Gamma(p + q)} \int_0^x t^{p-1} (1 - t)^{q-1} dt
\]

where \( \Gamma(\cdot) \) is the gamma function. The value of the distribution function \( I_x(p, q) \) is the probability that the random variable takes a value less than or equal to \( x \).
The integral in the expression above is called the *incomplete beta function* and is denoted by \( \beta_x(p, q) \). The constant in the expression is the reciprocal of the *beta function* (the incomplete function evaluated at one) and is denoted by \( \beta_x(p, q) \).

**Beta** uses the method of Bosten and Battiste (1974).

---

**Beta Distribution Function**

\[ \text{beta}(x, p, q) \]

- **x**: A `double` specifying the argument at which the function is to be evaluated.
- **pin**: A `double` specifying the first beta distribution parameter.
qin  A double specifying the second beta distribution parameter.

Returns — A double specifying the probability that a beta random variable takes on a
value less than or equal to x.

Binomial

static public double Binomial(int k, int n, double p)
Evaluates the binomial distribution function.

Method Binomial evaluates the distribution function of a binomial random variable with
parameters n and p. It does this by summing probabilities of the random variable taking
on the specific values in its range. These probabilities are computed by the recursive
relationship

\[ \Pr(X = j) = \frac{(n + 1 - j)p}{j(1-p)} \Pr(X = j - 1) \]

To avoid the possibility of underflow, the probabilities are computed forward from 0, if k
is not greater than n times p, and are computed backward from n, otherwise. The
smallest positive machine number, \( \varepsilon \), is used as the starting value for summing the
probabilities, which are rescaled by \((1 - p)^n \varepsilon\) if forward computation is performed and by
\(p^n \varepsilon\) if backward computation is done. For the special case of \( p = 0 \), Binomial is set to 1;
and for the case \( p = 1 \), Binomial is set to 1 if \( k = n \) and to 0 otherwise.

k  A int specifying the argument for which the binomial distribution function is to be
evaluated.

n  A int specifying the number of Bernoulli trials.

p  A double specifying the probability of success on each trial.

Returns — A double specifying the probability that a binomial random variable takes a
value less than or equal to k. This value is the probability that k or fewer successes occur
in n independent Bernoulli trials, each of which has a p probability of success.

BinomialProb

static public double BinomialProb(int k, int n, double p)
Evaluate the binomial probability function.

Method BinomialProb evaluates the probability that a binomial random variable with
parameters n and p takes on the value k. It does this by computing probabilities of the
random variable taking on the values in its range less than (or the values greater than) k.
These probabilities are computed by the recursive relationship

\[ \Pr(X = j) = \frac{(n + 1 - j)p}{j(1-p)} \Pr(X = j - 1) \]

To avoid the possibility of underflow, the probabilities are computed forward from 0, if k
is not greater than n times p, and are computed backward from n, otherwise. The smallest
positive machine number, \( \varepsilon \), is used as the starting value for computing the probabilities,
which are rescaled by \((1 - p)^n \varepsilon\) if forward computation is performed and by \(p^n \varepsilon\) if backward computation is done.

For the special case of \(p = 0\), BinomialProb is set to 0 if \(k\) is greater than 0 and to 1 otherwise; and for the case \(p = 1\), BinomialProb is set to 0 if \(k\) is less than \(n\) and to 1 otherwise.

\[\text{Binomial Probability Function}\]

\[\begin{array}{c|ccccccccccc}
0 & 0.00 & 0.10 & 0.20 & 0.30 & 0.40 \\
\end{array}\]

\[\text{Probablity} \]

\[\text{n=10 p=0.5} \quad n=10 p=0.2 \]

\[\text{k} \quad \text{A int specifying the argument for which the binomial distribution function is to be evaluated.}\]
n  A **int** specifying the number of Bernoulli trials.

p  A **double** specifying the probability of success on each trial.

Returns — A **double** specifying the probability that a binomial random variable takes a value equal to k.

Chi

```java
static public double Chi(double chsq, double df)
```

Evaluate the chi-squared distribution function.

Method **Chi** evaluates the distribution function, \( F \), of a chi-squared random variable with \( df \) degrees of freedom, that is, with \( v = df \), and \( x = chsq \).

\[
F(x) = \frac{1}{2^{\nu/2} \Gamma(\nu/2)} \int_0^x e^{-t/2\nu/2-1} \, dt
\]

where \( \Gamma(\cdot) \) is the gamma function. The value of the distribution function at the point \( x \) is the probability that the random variable takes a value less than or equal to \( x \).

For \( v > 65 \), **Chi** uses the Wilson-Hilferty approximation (Abramowitz and Stegun 1964, equation 26.4.17) to the normal distribution, and method **Normal** is used to evaluate the normal distribution function.

For \( v \leq 65 \), **Chi** uses series expansions to evaluate the distribution function. If \( x < \max(v/2, 26) \), **Chi** uses the series 6.5.29 in Abramowitz and Stegun (1964), otherwise, it uses the asymptotic expansion 6.5.32 in Abramowitz and Stegun.
Chi-Squared Distribution Function

F

\text{static public double } F(\text{double } x, \text{ double } dfn, \text{ double } dfd)

\text{chsq} \quad \text{A double specifying the argument at which the function is to be evaluated.}
\text{df} \quad \text{A double specifying the number of degrees of freedom. This must be at least 0.5.}

\text{Returns — A double specifying the probability that a chi-squared random variable takes a values less than or equal to chsq.}
Evaluates the F distribution function.

\( F \) evaluates the distribution function of a Snedecor’s F random variable with \( dfn \) numerator degrees of freedom and \( dfd \) denominator degrees of freedom. The function is evaluated by making a transformation to a beta random variable and then using the function \( \text{Beta} \). If \( X \) is an \( F \) variate with \( v_1 \) and \( v_2 \) degrees of freedom and \( Y = v_1 X / (v_2 + v_1 X) \), then \( Y \) is a beta variate with parameters \( p = v_1 / 2 \) and \( q = v_2 / 2 \). \( F \) also uses a relationship between \( F \) random variables that can be expressed as follows:

\[
F(X, dfn, dfd) = 1 - F(1/X, dfd, dfn)
\]
x  A double specifying the argument at which the function is to be evaluated.

dfn  A double specifying the numerator degrees of freedom. It must be positive.

dfd  A double specifying the denominator degrees of freedom. It must be positive.

Returns — A double specifying the probability that an F random variable takes on a value less than or equal to x.
Gamma

static public double Gamma(double x, double a)
Evaluates the gamma distribution function.

Method Gamma evaluates the distribution function, \( F \), of a gamma random variable with shape parameter \( a \); that is,

\[
F(x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt
\]

where \( \Gamma(\cdot) \) is the gamma function. (The gamma function is the integral from 0 to \( \infty \) of the same integrand as above). The value of the distribution function at the point \( x \) is the probability that the random variable takes a value less than or equal to \( x \).

The gamma distribution is often defined as a two-parameter distribution with a scale parameter \( b \) (which must be positive), or even as a three-parameter distribution in which the third parameter \( c \) is a location parameter. In the most general case, the probability density function over \((c, \infty)\) is

\[
f(t) = \frac{1}{b^a \Gamma(a)} e^{-(t-c)/b} (x-c)^{a-1}
\]

If \( T \) is such a random variable with parameters \( a, b, \) and \( c \), the probability that \( T \leq t_0 \) can be obtained from Gamma by setting \( X = (t_0 - c)/b \).

If \( X \) is less than \( a \) or if \( X \) is less than or equal to 1.0, Gamma uses a series expansion. Otherwise, a continued fraction expansion is used. (See Abramowitz and Stegun, 1964.)
A double specifying the argument at which the function is to be evaluated.

A double specifying the shape parameter. This must be positive.

Returns — A double specifying the probability that a gamma random variable takes on a value less than or equal to x.
The hypergeometric distribution function evaluates the probability of getting a certain number of successes in a sample drawn from a finite population without replacement. The parameters involved are:

- \( n \): The size of the finite population.
- \( l \): The number of successes in the finite population.
- \( m \): The number of failures in the finite population.
- \( j \): The number of successes in the sample.
- \( k \): The number of successes to be evaluated.

The probability function is given by:

\[
Pr(X = j) = \frac{\binom{m}{j} \binom{l-m}{n-j}}{\binom{m}{n}}
\]

for \( j = i, i+1, i+2, \ldots, \min(n,m) \)

where \( i = \max(0, n - l + m) \).

If \( k \) is greater than or equal to \( i \) and less than or equal to \( \min(n,m) \), Hypergeometric sums the terms in this expression for \( j \) going from \( i \) up to \( k \). Otherwise, 0 or 1 is returned, as appropriate. So, as to avoid rounding in the accumulation, Hypergeometric performs the summation differently depending on whether or not \( k \) is greater than the mode of the distribution, which is the greatest integer less than or equal to \( \frac{(m+1)(n+1)}{(l+2)} \).

**HypergeometricProb**

- **k** A **int** specifying the argument at which the function is to be evaluated.
- **sampleSize** A **int** specifying the sample size.
- **defectivesInLot** A **int** specifying the number of defectives in the lot.
- **lotSize** A **int** specifying the lot size.

Returns — A **double** specifying the probability that a hypergeometric random variable takes a value less than or equal to \( k \).
defectivesInLot  A int specifying the number of defectives in the lot.
lotSize  A int specifying the lot size.

Returns — A double specifying the probability that a hypergeometric random variable takes a value equal to k.

InverseBeta

static public double InverseBeta(double p, double pin, double qin)
Evaluates the inverse of the beta probability distribution function.

Method InverseBeta evaluates the inverse distribution function of a beta random variable with parameters pin and qin, that is, with \( P = p, \ p = pin, \) and \( q = qin \), it determines \( x \) (equal to InverseBeta \((p, pin, qin)\)), such that

\[
P = \frac{\Gamma(p)\Gamma(q)}{\Gamma(p+q)} \int_0^x t^{p-1} (1-t)^{q-1} \, dt
\]

where \( \Gamma(\cdot) \) is the gamma function. The probability that the random variable takes a value less than or equal to \( x \) is \( P \).

- p  A double specifying the probability for which the inverse of the beta CDF is to be evaluated.
- pin  A double specifying the first beta distribution parameter.
- qin  A double specifying the second beta distribution parameter.

Returns — A double specifying the probability that a beta random variable takes a value less than or equal to this value is \( p \).

InverseChi

static public double InverseChi(double p, double df)
Evaluate the inverse of the chi-squared distribution function.

Method InverseChi evaluates the inverse distribution function of a chi-squared random variable with \( df \) degrees of freedom, that is, with \( P = p \) and \( v = df \), it determines \( x \) (equal to InverseChi \((p, df)\)), such that

\[
P = \frac{1}{2^{\nu/2}\Gamma(\nu/2)} \int_0^x e^{-t/2}\nu/2 - 1 \, dt
\]

where \( \Gamma(\cdot) \) is the gamma function. The probability that the random variable takes a value less than or equal to \( x \) is \( P \).

For \( v < 40 \), InverseChi uses bisection, if \( v \geq 2 \) or \( P > 0.98 \), or regula falsi to find the point at which the chi-squared distribution function is equal to \( P \). The distribution function is evaluated using Chi.

For \( 40 \leq v < 100 \), a modified Wilson-Hilferty approximation (Abramowitz and Stegun 1964, equation 26.4.18) to the normal distribution is used, and InverseNormal is used to evaluate the inverse of the normal distribution function. For \( v \geq 100 \), the ordinary Wilson-Hilferty approximation (Abramowitz and Stegun 1964, equation 26.4.17) is used.
A double specifying the probability for which the inverse chi-squared function is to be evaluated.

*df* A double specifying the number of degrees of freedom. This must be at least 0.5.

Returns — A double specifying the probability that a chi-squared random variable takes a value less than or equal to this value is *p*.

**InverseF**

```java
static public double InverseF(double p, double dfn, double dfd)
```

Returns inverse of the F probability distribution function. Method **InverseF** evaluates the inverse distribution function of a Snedecor’s F random variable with *dfn* numerator degrees of freedom and *dfd* denominator degrees of freedom.

The function is evaluated by making a transformation to a beta random variable and then using **InverseBeta**. If *X* is an F variate with *v*₁ and *v*₂ degrees of freedom and *Y* = *v*₁*X*/(*v*₂ + *v*₁*X*), then *Y* is a beta variate with parameters *p* = *v*₁/2 and *q* = *v*₂/2. If *P* ≤ 0.5, **InverseF** uses this relationship directly, otherwise, it also uses a relationship between *X* random variables that can be expressed as follows, using *f*, which is the *F* cumulative distribution function:

\[ F(X, dfn, dfd) = 1 - F(1/X, dfd, dfn) \]

A double specifying the probability for which the inverse of the F distribution function is to be evaluated.

*dfn* A double specifying the numerator degrees of freedom. It must be positive.

*dfd* A double specifying the denominator degrees of freedom. It must be positive.

Returns — A double specifying the probability that an F random variable takes a value less than or equal to this value is *p*.

**InverseGamma**

```java
static public double InverseGamma(double p, double a)
```

Evaluates the inverse of the gamma distribution function. Method **InverseGamma** evaluates the inverse distribution function of a gamma random variable with shape parameter *a*, that is, it determines *x* (= **InverseGamma**(*p*, *a*)), such that

\[ P = \frac{1}{\Gamma(a)} \int_0^x e^{-t}t^{a-1}dt \]

where \( \Gamma(\cdot) \) is the gamma function. The probability that the random variable takes a value less than or equal to *x* is *P*. See the documentation for routine **Gamma** for further discussion of the gamma distribution.

**InverseGamma** uses bisection and modified regula falsi to invert the distribution function, which is evaluated using method **Gamma**.
Double specifying the probability at which the function is to be evaluated.

A double specifying the shape parameter. This must be positive.

Returns — A double specifying the probability that a gamma random variable takes a value less than or equal to this value is p.

**InverseNormal**

```csharp
static public double InverseNormal(double p)
```

Evaluates the inverse of the normal (Gaussian) distribution function.

Method `InverseNormal` evaluates the inverse of the distribution function, \( \Phi \), of a standard normal (Gaussian) random variable, that is, \( \text{InverseNormal}(p) = \Phi^{-1}(p) \), where

\[
\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt
\]

The value of the distribution function at the point \( x \) is the probability that the random variable takes a value less than or equal to \( x \). The standard normal distribution has a mean of 0 and a variance of 1.

**InverseStudentsT**

```csharp
static public double InverseStudentsT(double p, double df)
```

Returns inverse of the Student’s t distribution function.

Method `InverseStudentsT` evaluates the inverse distribution function of a Student’s t random variable with \( df \) degrees of freedom. Let \( v = df \). If \( v \) equals 1 or 2, the inverse can be obtained in closed form, if \( v \) is between 1 and 2, the relationship of a t to a beta random variable is exploited and `InverseBeta` is used to evaluate the inverse; otherwise the algorithm of Hill (1970) is used. For small values of \( v \) greater than 2, Hill’s algorithm inverts an integrated expansion in \( 1/(1 + t^2/v) \) of the t density. For larger values, an asymptotic inverse Cornish-Fisher type expansion about normal deviates is used.

**Normal**

```csharp
static public double Normal(double x)
```

Evaluates the normal (Gaussian) distribution function.

Method `Normal` evaluates the distribution function, \( \Phi \), of a standard normal (Gaussian) random variable, that is,
\[ \Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt \]

The value of the distribution function at the point \( x \) is the probability that the random variable takes a value less than or equal to \( x \).

The standard normal distribution (for which \texttt{Normal} is the distribution function) has mean of 0 and variance of 1. The probability that a normal random variable with mean \( \mu \) and variance \( \sigma^2 \) is less than \( y \) is given by \texttt{Normal} evaluated at \( (y - \mu)/\sigma \).

\( \Phi(x) \) is evaluated by use of the complementary error function, \texttt{erfc}. The relationship is:

\[ \Phi(x) = \text{erfc}(x/\sqrt{2.0})/2 \]
Normal Distribution Function

\[ \text{normal}(x) \]

- **x**  A double specifying the argument at which the function is to be evaluated.

Returns — A double specifying the probability that a normal variable takes a value less than or equal to \( x \).

Poisson

\[ \text{static public double Poisson(int k, double theta)} \]
Evaluates the Poisson distribution function.

**Poisson** evaluates the distribution function of a Poisson random variable with parameter \( \text{theta} \). \( \text{theta} \), which is the mean of the Poisson random variable, must be positive. The probability function (with \( \theta = \text{theta} \)) is

\[
f(x) = e^{-\theta} \frac{\theta^x}{x!} \text{ for } x = 0, 1, 2, \ldots
\]

The individual terms are calculated from the tails of the distribution to the mode of the distribution and summed. **Poisson** uses the recursive relationship

\[
f(x + 1) = f(x) \left( \frac{\theta}{x + 1} \right), \text{ for } x = 0, 1, 2, \ldots k - 1
\]

with \( f(0) = e^{-\theta} \).

- **k** A `int` specifying the argument for which the Poisson distribution function is to be evaluated.
- **theta** A `double` specifying the mean of the Poisson distribution.

Returns — A `double` specifying the probability that a Poisson random variable takes a value less than or equal to \( k \).

**PoissonProb**

```java
static public double PoissonProb(int k, double theta)
```

Evaluates the Poisson probability function.

Method **PoissonProb** evaluates the probability function of a Poisson random variable with parameter \( \text{theta} \). \( \text{theta} \), which is the mean of the Poisson random variable, must be positive. The probability function (with \( \theta = \text{theta} \)) is

\[
f(x) = e^{-\theta} \frac{\theta^k}{k!}, \text{ for } k = 0, 1, 2, \ldots
\]

**PoissonProb** evaluates this function directly, taking logarithms and using the log gamma function.
Poisson Probability Function

A **int** specifying the argument for which the Poisson probability function is to be evaluated.

**theta**  
A **double** specifying the mean of the Poisson distribution.

Returns — A **double** specifying the probability that a Poisson random variable takes a value equal to \( k \).
StudentsT

    static public double StudentsT(double t, double df)
Evaluates the Student’s t distribution function.

Method StudentsT evaluates the distribution function of a Student’s t random variable with df degrees of freedom. If the square of t is greater than or equal to df, the relationship of a t to an f random variable (and subsequently, to a beta random variable) is exploited, and routine Beta is used. Otherwise, the method described by Hill (1970) is used. If df is not an integer, if df is greater than 19, or if df is greater than 200, a Cornish-Fisher expansion is used to evaluate the distribution function. If df is less than 20 and |t| is less than 2.0, a trigonometric series (see Abramowitz and Stegun 1964, equations 26.7.3 and 26.7.4, with some rearrangement) is used. For the remaining cases, a series given by Hill (1970) that converges well for large values of t is used.
Student's t Distribution Function

![Student's t Distribution Function Graph](image_url)

\[ \text{students}(t, \nu) \]

\( \nu \)

- \( t \)  A **double** specifying the argument at which the function is to be evaluated.
- \( \text{df} \)  A **double** specifying the number of degrees of freedom. This must be at least one.

Returns — A **double** specifying the probability that a Student’s t random variable takes a value less than or equal to \( t \).
static public double Weibull(double x, double gamma, double alpha)
Evaluates the Weibull distribution function.

x  A double specifying the argument at which the function is to be evaluated. It must
be non-negative.

gamma  A double specifying the shape parameter.

alpha  A double specifying the scale parameter.

Returns — A double specifying the probability that a Weibull random variable takes a
value less than or equal to x.

Example: The Cumulative Distribution Functions

Various cumulative distribution functions are exercised. Their use in this example typifies the
manner in which other functions in the Cdf class would be used.

using System;
using Imsl.Stat;

public class CdfEx1
{
    public static void Main(String[] args)
    {
        double x, prob, result;
        int p, q, k, n;
        // Beta
        x = .5;
        p = 12;
        q = 12;
        result = Cdf.Beta(x, p, q);
        Console.Out.WriteLine("beta(.5, 12, 12) is " + result);

        // Inverse Beta
        x = .5;
        p = 12;
        q = 12;
        result = Cdf.InverseBeta(x, p, q);
        Console.Out.WriteLine("inversebeta(.5, 12, 12) is " + result);

        // binomial
        k = 3;
        n = 5;
        prob = .95;
        result = Cdf.Binomial(k, n, prob);
        Console.Out.WriteLine("binomial(3, 5, .95) is " + result);

        // Chi
        x = .15;
        n = 2;
        result = Cdf.Chi(x, n);
        Console.Out.WriteLine("chi(.15, 2) is " + result);
// Inverse Chi
prob = .99;
n = 2;
result = Cdf.InverseChi(prob, n);
Console.Out.WriteLine("inverseChi(.99, 2) is " + result);
}

Output

beta(.5, 12, 12) is 0.500000000000002
inversetbeta(.5, 12, 12) is 0.5
binomial(3, 5, .95) is 0.0225925
chi(.15, 2) is 0.0722565136714471
inverseChi(.99, 2) is 9.21034037197624

ICdfFunction Interface

Interface for the user-supplied cumulative distribution function to be used by InverseCdf and ChiSquaredTest.

public interface Imsl.Stat.ICdfFunction

Method

CdfFunction

abstract public double CdfFunction(double p)
User-supplied cumulative distribution function to be used by InverseCdf.

p A double scalar value representing the point at which the inverse CDF is desired.

Returns A double scalar value representing the probability that a random variable for this CDF takes a value less than or equal to this value is p.

InverseCdf Class

Inverse of user-supplied cumulative distribution function.

public class Imsl.Stat.InverseCdf
Property

Tolerance

public double Tolerance {get; set; }

The tolerance to be used as the convergence criterion. A double scalar value
representing the convergence criterion.

When the relative change from one iteration to the next is less than tolerance,
convergence is assumed. The default value for tolerance is 0.0001.

Constructor

InverseCdf

public InverseCdf(Imsl.Stat.ICdfFunction cdf)

Constructor for the inverse of a user-supplied cumulative distribution function.
The cdf function must be continuous and strictly monotone.

cdf  A ICdfFunction object that contains the user-supplied function to be inverted.

Method

Eval

public double Eval(double p, double guess)

Evaluates the inverse CDF function.

Cdf(InverseCdf) is "close" to p.

p  A double scalar value representing the point at which the inverse CDF is desired.
guess  A double scalar value representing an initial estimate of the inverse at p.

Returns — A double scalar value representing the inverse of the CDF at the point p.

Imsl.Stat.DidNotConvergeException is thrown if the iteration to find the inverse of
the CDF did not converge.

Description

Class InverseCdf evaluates the inverse of a continuous, strictly monotone function. Its most
obvious use is in evaluating inverses of continuous distribution functions that can be defined by
a user-supplied function, which implements the ICdfFunction interface. The inverse is
computed using regula falsi and/or bisection, possibly with the Illinois modification (see
Dahlquist and Bjorck 1974). A maximum of 100 iterations are performed.
Example: Inverse of a User-Supplied Cumulative Distribution Function

In this example, InverseCdf is used to compute the point such that the probability is 0.9 that a standard normal random variable is less than or equal to the computed point.

```csharp
using System;
using Imsl.Stat;

class InverseCdfEx1 : ICdfFunction
{
    public double CdfFunction(double x)
    {
        return Cdf.Normal(x);
    }

    public static void Main(string[] args)
    {
        double p = 0.9;
        ICdfFunction normal = new InverseCdfEx1();
        InverseCdf inv = new InverseCdf(normal);
        inv.Tolerance = 1.0e-10;
        double x1 = inv.Eval(p, 0.0);
        Console.WriteLine("The 90th percentile of a standard normal is " + x1);
    }
}
```

Output

The 90th percentile of a standard normal is 1.2815515655446
Chapter 21: Random Number Generation

Types

- class Random
- class FaureSequence
- interface IRandomSequence

Random Class

Generate uniform and non-uniform random number distributions.

public class Imsl.Stat.Random : Random

Property

Multiplier

public System.Int64 Multiplier {get; set; }

The multiplier for a linear congruential random number generator. A long which represents the random number generator multiplier.

If not set, the multiplier has the value zero. If a multiplier is set then the linear congruential generator, defined in the base class System.Random, is replaced by the generator

seed = (multiplier*seed) mod (2^{31} − 1)

See Donald Knuth, The Art of Computer Programming, Volume 2, for guidelines in choosing a multiplier. Some possible values are 16807, 397204094, 950706376.
Constructors

Random
public Random()
Constructor for the Random number generator class.

Random
public Random(int seed)
Constructor for the Random number generator class with supplied seed.

Seeds
A int which represents the random number generator seed.

Methods

Next
virtual public int Next(int maxValue)
Returns a nonnegative random number less than the specified maximum.

Next
virtual public int Next(int minValue, int maxValue)
Returns a random number within a specified range.

Next
virtual public int Next()
Returns a nonnegative random number.

NextBeta
virtual public double NextBeta(double p, double q)
Generate a pseudorandom number from a beta distribution.

Method NextBeta generates pseudorandom numbers from a beta distribution with
parameters p and q, both of which must be positive. The probability density function is

\[
f(x) = \frac{\Gamma(p + q)}{\Gamma(p)\Gamma(q)} x^{p-1} (1 - x)^{q-1} \quad \text{for} \ 0 \leq x \leq 1
\]

where \(\Gamma(\cdot)\) is the gamma function.

The algorithm used depends on the values of p and q. Except for the trivial cases of p = 1
or q = 1, in which the inverse CDF method is used, all of the methods use
acceptance/rejection. If p and q are both less than 1, the method of Johnk (1964) is used;
if either p or q is less than 1 and the other is greater than 1, the method of Atkinson
(1979) is used; if both $p$ and $q$ are greater than 1, algorithm BB of Cheng (1978), which requires very little setup time, is used.

The value returned is less than 1.0 and greater than $\varepsilon$, where $\varepsilon$ is the smallest positive number such that $1.0 - \varepsilon$ is less than 1.0.

- **p** A double which specifies the first beta distribution parameter, $p > 0$.
- **q** A double which specifies the second beta distribution parameter, $q > 0$.

Returns — A double which specifies a pseudorandom number from a beta distribution.

**NextBinomial**

```csharp
virtual public int NextBinomial(int n, double p)
```

Generate a pseudorandom number from a Binomial distribution.

**NextBinomial** generates pseudorandom numbers from a Binomial distribution with parameters $n$ and $p$. $n$ and $p$ must be positive, and $p$ must be less than 1. The probability function (with $n = n$ and $p = p$) is

$$f(x) = \binom{n}{x} p^x (1 - p)^{n-x}$$

for $x = 0, 1, 2, \ldots, n$.

The algorithm used depends on the values of $n$ and $p$. If $np < 10$ or if $p$ is less than a machine epsilon, the inverse CDF technique is used; otherwise, the BTPE algorithm of Kachitvichyanukul and Schmeiser (see Kachitvichyanukul 1982) is used. This is an acceptance/rejection method using a composition of four regions. (TPE equals Triangle, Parallelogram, Exponential, left and right.)

- **n** A int which specifies the number of Bernoulli trials.
- **p** A double which specifies the probability of success on each trial, $0 < p < 1$.

Returns — A int which specifies the pseudorandom number from a Binomial distribution.

**NextBytes**

```csharp
virtual public void NextBytes(System.Byte[] buffer)
```

Fills the elements of a specified array of bytes with random numbers.

**NextCauchy**

```csharp
virtual public double NextCauchy()
```

Generates a pseudorandom number from a Cauchy distribution.

The probability density function is

$$f(x) = \frac{1}{\pi(1 + x^2)}$$
Use of the inverse CDF technique would yield a Cauchy deviate from a uniform (0, 1) deviate, \( u \), as \( \tan[\pi(u - .5)] \). Rather than evaluating a tangent directly, however, NextCauchy generates two uniform (-1, 1) deviates, \( x_1 \) and \( x_2 \). These values can be thought of as sine and cosine values. If

\[
x_1^2 + x_2^2
\]

is less than or equal to 1, then \( x_1/x_2 \) is delivered as the Cauchy deviate; otherwise, \( x_1 \) and \( x_2 \) are rejected and two new uniform (-1, 1) deviates are generated. This method is also equivalent to taking the ratio of two independent normal deviates.

Deviates from the Cauchy distribution with median \( t \) and first quartile \( t - s \), that is, with density

\[
f(x) = \frac{s}{\pi [s^2 + (x - t)^2]}
\]

can be obtained by scaling the output from NextCauchy. To do this, first scale the output from NextCauchy by \( S \) and then add \( T \) to the result.

Returns — A double which specifies a pseudorandom number from a Cauchy distribution.

NextChiSquared

virtual public double NextChiSquared(double df)
Generates a pseudorandom number from a Chi-squared distribution.

NextChiSquared generates pseudorandom numbers from a chi-squared distribution with \( df \) degrees of freedom. If \( df \) is an even integer less than 17, the chi-squared deviate \( r \) is generated as

\[
r = -2 \ln\left( \prod_{i=1}^{n} u_i \right)
\]

where \( n = df/2 \) and the \( u_i \) are independent random deviates from a uniform (0, 1) distribution. If \( df \) is an odd integer less than 17, the chi-squared deviate is generated in the same way, except the square of a normal deviate is added to the expression above. If \( df \) is greater than 16 or is not an integer, and if it is not too large to cause overflow in the gamma random number generator, the chi-squared deviate is generated as a special case of a gamma deviate, using NextGamma. If overflow would occur in NextGamma, the chi-squared deviate is generated in the manner described above, using the logarithm of the product of uniforms, but scaling the quantities to prevent underflow and overflow.

\( df \) A double which specifies the number of degrees of freedom. It must be positive.

Returns — A double which specifies a pseudorandom number from a Chi-squared distribution.
NextDouble

```java
override public double NextDouble()
Generates the next pseudorandom number.
If the multiplier is set then the multiplicative congruential method is used. Otherwise, super.Next(bits) is used. Where bits is the number of random bits required.

Returns — A double which specifies the next pseudorandom value from this random number generator’s sequence.
```

NextExponential

```java
virtual public double NextExponential()
Generates a pseudorandom number from a standard exponential distribution.
The probability density function is \( f(x) = e^{-x}; \) for \( x > 0. \)
NextExponential uses an antithetic inverse CDF technique; that is, a uniform random deviate \( U \) is generated and the inverse of the exponential cumulative distribution function is evaluated at \( 1.0 - U \) to yield the exponential deviate.

Deviates from the exponential distribution with mean \( \text{THETA} \) can be generated by using NextExponential and then multiplying the result by \( \text{THETA}. \)

Returns — A double which specifies a pseudorandom number from a standard exponential distribution.
```

NextExponentialMix

```java
virtual public double NextExponentialMix(double theta1, double theta2, double p)
Generate a pseudorandom number from a mixture of two exponential distributions.
The probability density function is
\[
f(x) = \frac{p}{\theta_1} e^{-x/\theta_1} + \frac{1-p}{\theta_2} e^{-x/\theta_2} \text{ for } x > 0
\]
where \( p = p, \theta_1 = \text{theta1}, \) and \( \theta_2 = \text{theta2}. \)
In the case of a convex mixture, that is, the case \( 0 < p < 1, \) the mixing parameter \( p \) is interpretable as a probability; and NextExponentialMix with probability \( p \) generates an exponential deviate with mean \( \theta_1, \) and with probability \( 1 - p \) generates an exponential with mean \( \theta_2. \) When \( p \) is greater than \( 1, \) but less than \( \theta_1/(\theta_1 - \theta_2), \) then either an exponential deviate with mean \( \theta_2 \) or the sum of two exponentials with means \( \theta_1 \) and \( \theta_2 \) is generated. The probabilities are \( q = p - (p - 1)\theta_1/\theta_2 \) and \( 1 - q, \) respectively, for the single exponential and the sum of the two exponentials.

\textbf{theta1} A double which specifies the mean of the exponential distribution that has the larger mean.

\textbf{theta2} A double which specifies the mean of the exponential distribution that has the smaller mean. \textbf{theta2} must be positive and less than or equal to \textbf{theta1}.

\textbf{p} A double which specifies the mixing parameter. It must satisfy \( 0 \leq p \leq \text{theta1}/(\text{theta1} - \text{theta2}). \)
Returns — A double which specifies a pseudorandom number from a mixture of the two exponential distributions.

**NextFloat**

```csharp
public float NextFloat()
```
Generates the next pseudorandom number.

If the multiplier is set then the multiplicative congruential method is used. Otherwise, `super.Next(bits)` is used. Where bits is the number of random bits required.

Returns — A float which specifies the next pseudorandom value from this random number generator’s sequence.

**NextGamma**

```csharp
virtual public double NextGamma(double a)
```
Generates a pseudorandom number from a standard gamma distribution.

Method `NextGamma` generates pseudorandom numbers from a gamma distribution with shape parameter `a`. The probability density function is

\[
P = \frac{1}{\Gamma(a)} \int_0^x e^{-t}t^{a-1}dt
\]

Various computational algorithms are used depending on the value of the shape parameter `a`. For the special case of `a = 0.5`, squared and halved normal deviates are used; and for the special case of `a = 1.0`, exponential deviates (from method `NextExponential`) are used. Otherwise, if `a` is less than 1.0, an acceptance-rejection method due to Ahrens, described in Ahrens and Dieter (1974), is used; if `a` is greater than 1.0, a ten-region rejection procedure developed by Schmeiser and Lal (1980) is used.

The Erlang distribution is a standard gamma distribution with the shape parameter having a value equal to a positive integer; hence, `NextGamma` generates pseudorandom deviates from an Erlang distribution with no modifications required.

**a**  A double which specifies the shape parameter of the gamma distribution. It must be positive.

Returns — A double which specifies a pseudorandom number from a standard gamma distribution.

**NextGeometric**

```csharp
virtual public int NextGeometric(double p)
```
Generate a pseudorandom number from a geometric distribution.

`NextGeometric` generates pseudorandom numbers from a geometric distribution with parameter `p`, where \( P = p \) is the probability of getting a success on any trial. A geometric deviate can be interpreted as the number of trials until the first success (including the trial in which the first success is obtained). The probability function is

\[
f(x) = P(1 - P)^{x-1}
\]
for $x = 1, 2, \ldots$ and $0 < P < 1$.

The geometric distribution as defined above has mean $1/P$.

The $i$-th geometric deviate is generated as the smallest integer not less than \( \log(U_i)/\log(1 - P) \), where the $U_i$ are independent uniform (0, 1) random numbers (see Knuth, 1981).

The geometric distribution is often defined on 0, 1, 2, ..., with mean $(1 - P)/P$. Such deviates can be obtained by subtracting 1 from each element returned value.

- $p$ A double which specifies the probability of success on each trial, $0 < p \leq 1$.

Returns — A int which specifies a pseudorandom number from a geometric distribution.

NextHypergeometric

virtual public int NextHypergeometric(int n, int m, int l)

Generate a pseudorandom number from a hypergeometric distribution.

Method NextHypergeometric generates pseudorandom numbers from a hypergeometric distribution with parameters $n$, $m$, and $l$. The hypergeometric random variable $x$ can be thought of as the number of items of a given type in a random sample of size $n$ that is drawn without replacement from a population of size $l$ containing $m$ items of this type. The probability function is

\[
 f(x) = \frac{\binom{m}{x} \binom{l-m}{n-x}}{\binom{l}{n}}
\]

for $x = \max(0, n - l + m), 1, 2, \ldots, \min(n, m)$.

If the hypergeometric probability function with parameters $n$, $m$, and $l$ evaluated at $n - l + m$ (or at 0 if this is negative) is greater than the machine epsilon, and less than 1.0 minus the machine epsilon, then NextHypergeometric uses the inverse CDF technique. The method recursively computes the hypergeometric probabilities, starting at $x = \max(0, n - l + m)$ and using the ratio $f(x + 1)/f(x)$ (see Fishman 1978, page 457).

If the hypergeometric probability function is too small or too close to 1.0, then NextHypergeometric generates integer deviates uniformly in the interval $[1, l - i]$, for $i = 0, 1, \ldots$; and at the $I$-th step, if the generated deviate is less than or equal to the number of special items remaining in the lot, the occurrence of one special item is tallied and the number of remaining special items is decreased by one. This process continues until the sample size or the number of special items in the lot is reached, whichever comes first. This method can be much slower than the inverse CDF technique. The timing depends on $n$. If $n$ is more than half of $l$ (which in practical examples is rarely the case), the user may wish to modify the problem, replacing $n$ by $l - n$, and to consider the deviates to be the number of special items not included in the sample.

- $n$ A int which specifies the number of items in the sample, $n > 0$.
- $m$ A int which specifies the number of special items in the population, or lot, $m > 0$. 
A **int** which specifies the number of items in the lot, \( l > \max(n, m) \).

**Returns** — A **int** which specifies the number of special items in a sample of size \( n \) drawn without replacement from a population of size \( l \) that contains \( m \) such special items.

**NextLogNormal**

```csharp
virtual public double NextLogNormal(double mean, double stdev)
```

Generate a pseudorandom number from a lognormal distribution.

Method **NextLogNormal** generates pseudorandom numbers from a lognormal distribution with parameters **mean** and **stdev**. The scale parameter in the underlying normal distribution, **stdev**, must be positive. The method is to generate normal deviates with mean **mean** and standard deviation **stdev** and then to exponentiate the normal deviates.

With \( \mu = \text{mean} \) and \( \sigma = \text{stdev} \), the probability density function for the lognormal distribution is

\[
 f(x) = \frac{1}{\sigma x \sqrt{2\pi}} \exp \left( -\frac{1}{2\sigma^2} (\ln x - \mu)^2 \right) \text{ for } x > 0
\]

The mean and variance of the lognormal distribution are \( \exp(\mu + \sigma^2/2) \) and \( \exp(2\mu + 2\sigma^2) - \exp(2\mu + \sigma^2) \), respectively.

**mean** A **double** which specifies the mean of the underlying normal distribution.

**stdev** A **double** which specifies the standard deviation of the underlying normal distribution. It must be positive.

**Returns** — A **double** which specifies a pseudorandom number from a lognormal distribution.

**NextLogarithmic**

```csharp
virtual public int NextLogarithmic(double a)
```

Generate a pseudorandom number from a logarithmic distribution.

Method **NextLogarithmic** generates pseudorandom numbers from a logarithmic distribution with parameter **a**. The probability function is

\[
 f(x) = -\frac{a^x}{x \ln(1-a)}
\]

for \( x = 1, 2, 3, \ldots \), and \( 0 < a < 1 \).

The methods used are described by Kemp (1981) and depend on the value of \( a \). If \( a \) is less than 0.95, Kemp’s algorithm LS, which is a “chop-down” variant of an inverse CDF technique, is used. Otherwise, Kemp’s algorithm LK, which gives special treatment to the highly probable values of 1 and 2, is used.

**a** A **double** which specifies the parameter of the logarithmic distribution, \( 0 < a \leq 1 \).

**Returns** — A **int** which specifies a pseudorandom number from a logarithmic distribution.
NextMultivariateNormal

virtual public double[] NextMultivariateNormal(int k, Imsl.Math.Cholesky matrix)
Generate pseudorandom numbers from a multivariate normal distribution.

NextMultivariateNormal generates pseudorandom numbers from a multivariate normal distribution with mean vector consisting of all zeroes and variance-covariance matrix whose Cholesky factor (or ”square root”) is matrix; that is, matrix is an upper triangular matrix such that the transpose of matrix times matrix is the variance-covariance matrix.

First, independent random normal deviates with mean 0 and variance 1 are generated, and then the matrix containing these deviates is post-multiplied by matrix.

Deviates from a multivariate normal distribution with means other than zero can be generated by using NextMultivariateNormal and then by adding the means to the deviates.

k  A int which specifies the length of the multivariate normal vectors.
matrix  The Cholesky factorization of the variance-covariance matrix of order k.

Returns — A double array which contains the pseudorandom numbers from a multivariate normal distribution.

NextNegativeBinomial

virtual public int NextNegativeBinomial(double rk, double p)
Generate a pseudorandom number from a negative Binomial distribution.

Method NextNegativeBinomial generates pseudorandom numbers from a negative Binomial distribution with parameters rk and p. rk and p must be positive and p must be less than 1. The probability function with \(r = rk\) and \(p = p\) is

\[ f(x) = \binom{r + x - 1}{x} (1 - p)^r p^x \]

for \(x = 0, 1, 2, \ldots\).

If \(rp/(1 - p)\) is less than 100 and \((1 - p)^r\) is greater than the machine epsilon, NextNegativeBinomial uses the inverse CDF technique; otherwise, for each negative binomial deviate, NextNegativeBinomial generates a gamma \((r, p/(1 - p))\) deviate \(y\) and then generates a Poisson deviate with parameter \(y\).  

rk  A double which specifies the negative binomial parameter, \(rk > 0\).
p  A double which specifies the probability of success on each trial. It must be greater than machine precision and less than one.
Returns — A **int** which specifies the pseudorandom number from a negative binomial distribution. If \( rk \) is an integer, the deviate can be thought of as the number of failures in a sequence of Bernoulli trials before \( rk \) successes occur.

NextNormal

```csharp
virtual public double NextNormal()
```

Generate a pseudorandom number from a standard normal distribution using an inverse CDF method.

In this method, a uniform \((0,1)\) random deviate is generated, then the inverse of the normal distribution function is evaluated at that point using `InverseNormal`. This method is slower than the acceptance/rejection technique used in `NextNormalAR` to generate standard normal deviates. Deviates from the normal distribution with mean \( x_m \) and standard deviation \( x_{std} \) can be obtained by scaling the output from `NextNormal`. To do this first scale the output of `NextNormal` by \( x_{std} \) and then add \( x_m \) to the result.

Returns — A **double** which represents a pseudorandom number from a standard normal distribution.

NextNormalAR

```csharp
virtual public double NextNormalAR()
```

Generate a pseudorandom number from a standard normal distribution using an acceptance/rejection method.

`NextNormalAR` generates pseudorandom numbers from a standard normal (Gaussian) distribution using an acceptance/rejection technique due to Kinderman and Ramage (1976). In this method, the normal density is represented as a mixture of densities over which a variety of acceptance/rejection methods due to Marsaglia (1964), Marsaglia and Bray (1964), and Marsaglia, MacLaren, and Bray (1964) are applied. This method is faster than the inverse CDF technique used in `NextNormal` to generate standard normal deviates.

Deviates from the normal distribution with mean \( x_m \) and standard deviation \( x_{std} \) can be obtained by scaling the output from `NextNormalAR`. To do this first scale the output of `NextNormalAR` by \( x_{std} \) and then add \( x_m \) to the result.

Returns — A **double** which represents a pseudorandom number from a standard normal distribution.

NextPoisson

```csharp
virtual public int NextPoisson(double theta)
```

Generate a pseudorandom number from a Poisson distribution.

Method `NextPoisson` generates pseudorandom numbers from a Poisson distribution with parameter `theta`. `theta`, which is the mean of the Poisson random variable, must be positive. The probability function (with \( \theta = \text{theta} \)) is

\[
f(x) = e^{-\theta} \frac{\theta^x}{x!}
\]

for \( x = 0, 1, 2, \ldots \)
If theta is less than 15, NextPoisson uses an inverse CDF method; otherwise the PTPE method of Schmeiser and Kachitvichyanukul (1981) (see also Schmeiser 1983) is used. The PTPE method uses a composition of four regions, a triangle, a parallelogram, and two negative exponentials. In each region except the triangle, acceptance/rejection is used. The execution time of the method is essentially insensitive to the mean of the Poisson.

theta A double which specifies the mean of the Poisson distribution, theta > 0.

Returns — A int which specifies a pseudorandom number from a Poisson distribution.

NextStudentsT
virtual public double NextStudentsT(double df)
Generate a pseudorandom number from a Student’s t distribution.

NextStudentsT generates pseudo-random numbers from a Student’s t distribution with df degrees of freedom, using a method suggested by Kinderman, Monahan, and Ramage (1977). The method (“TMX” in the reference) involves a representation of the t density as the sum of a triangular density over (-2, 2) and the difference of this and the t density. The mixing probabilities depend on the degrees of freedom of the t distribution. If the triangular density is chosen, the variate is generated as the sum of two uniforms; otherwise, an acceptance/rejection method is used to generate a variate from the difference density.

For degrees of freedom less than 100, NextStudentsT requires approximately twice the execution time as NextNormalAR, which generates pseudorandom normal deviates. The execution time of NextStudentsT increases very slowly as the degrees of freedom increase. Since for very large degrees of freedom the normal distribution and the t distribution are very similar, the user may find that the difference in the normal and the t does not warrant the additional generation time required to use NextStudentsT instead of NextNormalAR.

df A double which specifies the number of degrees of freedom. It must be positive.

Returns — A double which specifies a pseudorandom number from a Student’s t distribution.

NextTriangular
virtual public double NextTriangular()
Generate a pseudorandom number from a triangular distribution on the interval (0,1).

The probability density function is \( f(x) = 4x \), for \( 0 \leq x \leq .5 \), and \( f(x) = 4(1 - x) \), for \( .5 < x \leq 1 \). NextTriangular uses an inverse CDF technique.

Returns — A double which specifies a pseudorandom number from a triangular distribution on the interval (0,1).

NextVonMises
virtual public double NextVonMises(double c)
Generate a pseudorandom number from a von Mises distribution.

Method NextVonMises generates pseudorandom numbers from a von Mises distribution with parameter c, which must be positive. With \( c = C \), the probability density function is
\[ f(x) = \frac{1}{2\pi I_0(c)} \exp[c \cos(x)] \text{ for } -\pi < x < \pi \]

where \(I_0(c)\) is the modified Bessel function of the first kind of order 0. The probability density equals 0 outside the interval \((-\pi, \pi)\).

The algorithm is an acceptance/rejection method using a wrapped Cauchy distribution as the majorizing distribution. It is due to Best and Fisher (1979).

\[ c \text{ A double which specifies the parameter of the von Mises distribution, } p > 7.4e - 9. \]

Returns — A double which specifies a pseudorandom number from a von Mises distribution.

NextWeibull

virtual public double NextWeibull(double a)
Generate a pseudorandom number from a Weibull distribution.

Method NextWeibull generates pseudorandom numbers from a Weibull distribution with shape parameter \(a\). The probability density function is

\[ f(x) = Ax^{A-1}e^{-x^A} \text{ for } x \geq 0 \]

NextWeibull uses an antithetic inverse CDF technique to generate a Weibull variate; that is, a uniform random deviate \(U\) is generated and the inverse of the Weibull cumulative distribution function is evaluated at \(1.0 - u\) to yield the Weibull deviate.

Deviates from the two-parameter Weibull distribution with shape parameter \(a\) can be generated by using NextWeibull and then multiplying the result by \(b\).

The Rayleigh distribution with probability density function,

\[ r(x) = \frac{1}{\alpha^2}xe^{(-x^2/2\alpha^2)} \text{ for } x \geq 0 \]

is the same as a Weibull distribution with shape parameter \(a\) equal to 2 and scale parameter \(b\) equal to

\[ \sqrt{2\alpha} \]

hence, NextWeibull and simple multiplication can be used to generate Rayleigh deviates.

\[ a \text{ A double which specifies the shape parameter of the Weibull distribution, } a > 0. \]

Returns — A double which specifies a pseudorandom number from a Weibull distribution.
Sample
    virtual double Sample()
    Returns a random number between 0.0 and 1.0.

Skip
    virtual public void Skip(int n)
    Resets the seed to skip ahead in the base linear congruential generator.

    This method can be used only if a linear congruential multiplier is explicitly defined by a

    The method skips ahead in the deviates returned by the protected method Random.Next.
The public methods use Next(int) as their source of uniform random deviates. Some
methods call it more than once. For instance, each call to Imsl.Stat.Random.NextDouble
(p. 578) calls it twice.

    n  A int which specifies the number of random deviates to skip.

Description

The non-uniform distributions are generated from a uniform distribution. By default, this class
uses the uniform distribution generated by the base class System.Random. If the multiplier is
set in this class then a multiplicative congruential method is used. The form of the generator is

\[ x_i \equiv cx_{i-1}\mod(2^{31} - 1) \]

Each \( x_i \) is then scaled into the unit interval \((0,1)\). If the multiplier, \( c \), is a primitive root
modulo \( 2^{31} - 1 \) (which is a prime), then the generator will have a maximal period of \( 2^{31} - 2 \).

There are several other considerations, however. See Knuth (1981) for a good general
discussion. Possible values for \( c \) are 16807, 397204094, and 950706376. The selection is made by
the property Imsl.Stat.Random.Multiplier (p. 575). Evidence suggests that the performance of
950706376 is best among these three choices (Fishman and Moore 1982).

The generation of uniform \((0,1)\) numbers is done by the method Imsl.Stat.Random.NextFloat
(p. 580).

Example: Random Number Generation

In this example, a discrete normal random sample of size 1000 is generated via NextNormal.
After the ChiSquaredTest constructor is called, the random observations are added to the test
one at a time to simulate streaming data. The Chi-squared test is performed using Cdf.Normal
as the cumulative distribution function object to see how well the random numbers fit the
normal distribution.

using System;
using Imsl.Stat;

public class RandomEx1 : ICdfFunction
{

Random Number Generation
Random Class • 587
public double CdfFunction(double x)
{
    return Cdf.Normal(x);
}

public static void Main(String[] args)
{
    int nObservations = 1000;
    ICdfFunction normal = new RandomEx1();
    ChiSquaredTest test = new ChiSquaredTest(normal, 10, 0);
    for (int k = 0; k < nObservations; k++)
    {
        test.Update(r.NextNormal(), 1.0);
    }
    double p = test.P;
    Console.Out.WriteLine("The P-value is " + p);
}

Output
The P-value is 0.496307043723263

FaureSequence Class

Generates the low-discrepancy Faure sequence.

public class Imsl.Stat.FaureSequence implements Properties

Base
    public int Base {get; }
    The base. A int which specifies the base.

Dimension
    Final public int Dimension {get; }
    Returns the dimension of the sequence. A int which specifies the dimension.

Skip
    public int Skip {get; }
    Returns the number of points skipped at the beginning of the sequence. A int which specifies the number of points skipped.
Constructors

FaureSequence

public FaureSequence(int dimension)
Creates a Faure sequence with the default base.
The base defaults to the smallest prime equal to or greater than dimension.

dimension  An int which specifies the dimension of the sequence.

FaureSequence

public FaureSequence(int dimension, int baseSequence, int nSkip)
Creates a Faure sequence.
If nSkip is negative then base^{m/2−1}, where m is the number of digits needed to represent
the largest Int32 in the base, points are skipped.

dimension  An int which specifies the dimension of the sequence.
baseSequence  A int which specifies the smallest prime number greater than or equal
to dimension.
nSkip  An int which specifies the number of initial points to skip.

Methods

ComputeParameters

void ComputeParameters()
Compute needed parameters.

NextDouble

public double NextDouble()
Returns the first value of the next point in the sequence.
This method is intended for use when dimension is 1.
Returns — A double array which specifies the next sequence value.

NextPoint

Final public double[] NextPoint()
Returns the next point in the sequence.
Returns — A double array which specifies the next point in the sequence.

NextPrime

static public int NextPrime(int n)
Returns the smallest prime greater than or equal to n.
If n is less than or equal to 2 then 2 is returned.

n  An int which specifies the first number to try as a prime.
Returns — An int which specifies a prime greater than or equal to n.
Description

Discrepancy measures the deviation from uniformity of a point set.
The discrepancy of the point set \( x_1, \ldots, x_n \in [0, 1]^d, \ d \geq 1, \) is
\[
D_n^{(d)} = \sup_E \left| \frac{A(E; n)}{n} - \lambda(E) \right|
\]
where the supremum is over all subsets of \([0, 1]^d\) of the form
\[
E = [0, t_1) \times \cdots \times [0, t_d), \ 0 \leq t_j \leq 1, \ 1 \leq j \leq d,
\]
\( \lambda \) is the Lebesque measure, and \( A(E; n) \) is the number of the \( x_j \) contained in \( E \).
The sequence \( x_1, x_2, \ldots \) of points in \([0, 1]^d\) is a low-discrepancy sequence if there exists a constant \( c(d) \), depending only on \( d \), such that
\[
D_n^{(d)} \leq c(d) \left( \frac{\log n}{n} \right)^d
\]
for all \( n > 1 \).

Generalized Faure sequences can be defined for any prime base \( b \geq d \). The lowest bound for the discrepancy is obtained for the smallest prime \( b \geq d \), so the base defaults to the smallest prime greater than or equal to the dimension.
The generalized Faure sequence \( x_1, x_2, \ldots \), is computed as follows:
Write the positive integer \( n \) in its \( b \)-ary expansion,
\[
n = \sum_{i=0}^{\infty} a_i(n)b^i
\]
where \( a_i(n) \) are integers, \( 0 \leq a_j(n) < b \).
The \( j \)-th coordinate of \( x_n \) is
\[
x_n^{(j)} = \sum_{k=0}^{\infty} \sum_{d=0}^{\infty} c_{kd}^{(j)} a_d(n)b^{-k-1}, \ 1 \leq j \leq d
\]
The generator matrix for the series, \( c_{kd}^{(j)} \), is defined to be
\[
c_{kd}^{(j)} = j^{d-k} c_{kd}
\]
and \( c_{kd} \) is an element of the Pascal matrix,
\[
c_{kd} = \left\{ \begin{array}{ll}
\frac{d!}{c_k(d-k)!} & k \leq d \\
0 & k > d
\end{array} \right.
\]
It is faster to compute a shuffled Faure sequence than to compute the Faure sequence itself. It can be shown that this shuffling preserves the low-discrepancy property.

The shuffling used is the $b$-ary Gray code. The function $G(n)$ maps the positive integer $n$ into the integer given by its $b$-ary expansion. The sequence computed by this function is $\vec{x}(G(n))$, where $\vec{x}$ is the generalized Faure sequence.

**Example: FaureSequence**

In this example, ten points of the Faure sequence are computed. The points are in a four-dimensional cube.

```csharp
using System;
using FaureSequence = Imsl.Stat.FaureSequence;
using PrintMatrix = Imsl.Math.PrintMatrix;
public class FaureSequenceEx1
{
    public static void Main(String[] args)
    {
        FaureSequence seq = new FaureSequence(4);
        double[][] x = new double[10][];
        for (int k = 0; k < 10; k++)
        {
            x[k] = seq.NextPoint();
        }
        new PrintMatrix("Faure Sequence").Print(x);
    }
}
```

**Output**

```
            0  1  2  3
 0 0.201344 0.274944 0.532544 0.694144
 1 0.401344 0.474944 0.732544 0.894144
 2 0.601344 0.674944 0.932544 0.094144
 3 0.801344 0.874944 0.132544 0.294144
 4 0.841344 0.114944 0.572544 0.934144
 5 0.041344 0.314944 0.772544 0.134144
 6 0.241344 0.514944 0.972544 0.334144
 7 0.441344 0.714944 0.172544 0.534144
 8 0.641344 0.914944 0.372544 0.734144
 9 0.681344 0.154944 0.612544 0.374144
```
IRandomSequence Interface

Interface implemented by generators of random or quasi-random multidimension sequences.

```csharp
public interface Imsl.Stat.IRandomSequence
```

**Property**

Dimension

```csharp
abstract public int Dimension {get; }
```

Returns the dimension of the sequence. A `int` which specifies the dimension.

**Method**

NextPoint

```csharp
abstract public double[] NextPoint()
```

Returns the next multidimensional point in the sequence.

Returns — A `double` array of length `dimension`.
Chapter 22: Finance

Types

class Finance ................................................................. 594
enumeration Finance.Period ............................................... 623
interface IBasisPart ......................................................... 623
class Bond ................................................................. 624
enumeration Bond.Frequency ........................................... 656
class DayCountBasis ....................................................... 657

Usage Notes

Users can perform financial computations by using pre-defined data types. Most of the financial functions require one or more of the following:

- Date
- Number of payments per year
- A variable to indicate when payments are due
- Day count basis

The Bond.Frequency field indicates the number of payments for each year.

<table>
<thead>
<tr>
<th>Bond.Frequency</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bond.Annual</td>
<td>One payment per year (Annual payment)</td>
</tr>
<tr>
<td>Bond.SemiAnnual</td>
<td>Two payments per year (Semi-annual payment)</td>
</tr>
<tr>
<td>Bond.Quarterly</td>
<td>Four payments per year (Quarterly payment)</td>
</tr>
</tbody>
</table>

The Finance.Period field indicates when payments are due.

<table>
<thead>
<tr>
<th>Finance.Period</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finance.At_End_of_Period</td>
<td>Payments are due at the end of the period</td>
</tr>
<tr>
<td>Finance.AT_Beginning_of_Period</td>
<td>Payments are due at the beginning of the period</td>
</tr>
</tbody>
</table>
The `DayCountBasis` class provides fields to indicate the type of day count basis. Day count basis is the method for computing the number of days between two dates.

<table>
<thead>
<tr>
<th>Class Field</th>
<th>Day count basis</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DayCountBasis.BasisNASD</code></td>
<td>US (NASD) 30/360</td>
</tr>
<tr>
<td><code>DayCountBasis.BasisActualActual</code></td>
<td>Actual/Actual</td>
</tr>
<tr>
<td><code>DayCountBasis.BasisActual360</code></td>
<td>Actual/360</td>
</tr>
<tr>
<td><code>DayCountBasis.BasisActual365</code></td>
<td>Actual/365</td>
</tr>
<tr>
<td><code>DayCountBasis.Basis30e360</code></td>
<td>European 30/360</td>
</tr>
</tbody>
</table>

**Additional Information**

In preparing the finance and bond functions we incorporated standards used by *SIA Standard Securities Calculation Methods*.

More detailed information on finance and bond functionality can be found in the following manuals:

- *Microsoft Excel 5, Worksheet Function Reference*.

---

**Finance Class**

Collection of finance functions.

```csharp
public class Imsl.Finance.Finance
```

**Constructor**

```csharp
public Finance()
```


**Methods**

```csharp
static public double Cumipmt(double rate, int nper, double pv, int firstPeriod, int lastPeriod, Imsl.Finance.Finance.Period period)
```

Returns the cumulative interest paid between two periods.

It is computed using the following:
\[ \sum_{i=firstPeriod}^{lastPeriod} interest_i \]

where \( interest_i \) is computed from \texttt{Ipmt} for the \( i \)-th period.

- **rate** A double which specifies the interest rate.
- **nper** A int which specifies the total number of payment periods.
- **pv** A double which specifies the present value.
- **firstPeriod** A int containing the first period in the calculation. Periods are numbered starting with one.
- **lastPeriod** A int which specifies the last period in the calculation.
- **period** A int which specifies the time in each period when the payment is made, either \texttt{Imsl.Finance.Finance.Period.AtEnd} (p. 623) or \texttt{Imsl.Finance.Finance.Period.AtBeginning} (p. 623)

Returns — A double which specifies the cumulative interest paid between the first period and the last period.

**Cumprinc**

\[
\sum_{i=firstPeriod}^{lastPeriod} principal_i
\]

where \( principal_i \) is computed from \texttt{Ppmt} for the \( i \)-th period.

- **rate** A double which specifies the interest rate.
- **nper** A int which specifies the total number of payment periods.
- **pv** A double which specifies the present value.
- **firstPeriod** A int which specifies the first period in the calculation. Periods are numbered starting with one.
- **lastPeriod** A int which specifies the last period in the calculation.
- **time** The time of a \texttt{Period} when the payment is made (either \texttt{Imsl.Finance.Finance.Period.AtEnd} (p. 623) or \texttt{Imsl.Finance.Finance.Period.AtBeginning} (p. 623)).

Returns — A double which specifies the cumulative principal paid between the first period and the last period.
Db

static public double Db(double cost, double salvage, int life, int period, int month)

Returns the depreciation of an asset using the fixed-declining balance method.

Method Db varies depending on the specified value for the argument period, see table below.

If period = 1,

\[ \text{cost} \times \text{rate} \times \frac{\text{month}}{12} \]

If period = life,

\[ (\text{cost} - \text{total depreciation from periods}) \times \text{rate} \times \frac{12 - \text{month}}{12} \]

If period other than 1 or life,

\[ (\text{cost} - \text{total depreciation from prior periods}) \times \text{rate} \]

where

\[ \text{rate} = 1 - \left( \frac{\text{salvage}}{\text{cost}} \right)^{\frac{1}{\text{life}}} \]

NOTE: rate is rounded to three decimal places.

cost  A double which specifies the initial cost of the asset.
salvage  A double which specifies the salvage value of the asset.
life  A int which specifies the number of periods over which the asset is being depreciated.
period  A int which specifies the period for which the depreciation is to be computed.
month  A int which specifies the number of months in the first year.

Returns — A double which specifies the depreciation of an asset for a specified period using the fixed-declining balance method.

Ddb

static public double Ddb(double cost, double salvage, int life, int period, double factor)

Returns the depreciation of an asset using the double-declining balance method.

It is computed using the following:

\[ \frac{[\text{cost} - \text{salvage (total depreciation from prior periods)}]}{\text{life}} \times \frac{\text{factor}}{} \]
cost  A double which specifies the initial cost of the asset.
salvage  A double which specifies the salvage value of the asset.
life  A int which specifies the number of periods over which the asset is being depreciated.
period  A int which specifies the period.
factor  A double which specifies the rate at which the balance declines.

Returns — A double which specifies the depreciation of an asset for a specified period.

Dollarde
static public double Dollarde(double fractionalDollar, int fraction)
Converts a fractional price to a decimal price.

It is computed using the following:

\[ idollar + (fractionalDollar - idollar) \times \frac{\frac{10^{(frac+1)}}{fraction}} \]

where idollar is the integer part of fractionalDollar, and ifrac is the integer part of \( \log(fraction) \).

fractionalDollar  A double which specifies a fractional number.
fraction  A int which specifies the denominator.

Returns — A double which specifies the dollar price expressed as a decimal number.

Dollarfr
static public double Dollarfr(double decimalDollar, int fraction)
Converts a decimal price to a fractional price.

It is computed using the following:

\[ idollar + \frac{decimalDollar - idollar}{\frac{10^{(frac+1)}}{fraction}} \]

where idollar is the integer part of the decimalDollar, and ifrac is the integer part of \( \log(fraction) \).

decimalDollar  A double which specifies a decimal number.
fraction  A int which specifies the denominator.

Returns — A double which specifies a dollar price expressed as a fraction.

Effect
static public double Effect(double nominalRate, int nper)
Returns the effective annual interest rate.

The nominal interest rate is the periodically-compounded interest rate as stated on the face of a security. The effective annual interest rate is computed using the following:
\[
(1 + \frac{\text{nominalRate}}{\text{nper}})^{\text{nper}} - 1
\]

**nominalRate**  A double which specifies the nominal interest rate.

**nper**  A int which specifies the number of compounding periods per year.

Returns — A double which specifies the effective annual interest rate.

---

**Fv**

```csharp
static public double Fv(double rate, int nper, double pmt, double pv,
Imsl.Finance.Finance.Period period)
```

Returns the future value of an investment.

The future value is the value, at some time in the future, of a current amount and a stream of payments. It can be found by solving the following:

If \( rate = 0 \),

\[
\text{pv} + \text{pmt} \times \text{nper} + \text{fv} = 0
\]

If \( rate \neq 0 \),

\[
\text{pv}(1 + rate)^{\text{nper}} + \text{pmt}[1 + rate(\text{period})] \left(\frac{(1 + rate)^{\text{nper}} - 1}{rate}\right) + \text{fv} = 0
\]

**rate**  A double which specifies the interest rate.

**nper**  A int which specifies the total number of payment periods.

**pmt**  A double which specifies the payment made in each period.

**pv**  A double which specifies the present value.

**period**  A int which specifies the time in each period when the payment is made (either Imsl.Finance.Finance.Period.AtEnd (p. 623) or Imsl.Finance.Finance.Period.AtBeginning (p. 623)).

Returns — A double which specifies the future value of an investment.

---

**Fvschedule**

```csharp
static public double Fvschedule(double principal, double[] schedule)
```

Returns the future value of an initial principal taking into consideration a schedule of compound interest rates.

It is computed using the following:

\[
\sum_{i=1}^{\text{count}} (\text{principal} \times \text{schedule}_i)
\]

where \( \text{schedule}_i = \) interest rate at the \( i \)-th period.
principal  A double which specifies the present value.
schedule  A double array of interest rates to apply.

Returns — A double which specifies the future value of an initial principal

Ipmt

\[ \text{Ipmt} \quad \text{static public double Ipmt(double rate, int period, int nper, double pv, double fv, Imsl.Finance.Finance.Period time)} \]

Returns the interest payment for an investment for a given period.

It is computed using the following:

\[
\left\{ \frac{pv (1 + rate)^{nper-1} + pmt (1 + rate \times period)}{rate} \right\} rate
\]

rate  A double which specifies the interest rate.
period  A int which specifies the payment period.
nper  A int which specifies the total number of periods.
pv  A double which specifies the present value.
fv  A double which specifies the future value.
time  The time of a Period when the payment is made (either 
Imsl.Finance.Finance.Period.AtEnd (p. 623) or 
Imsl.Finance.Finance.Period.AtBeginning (p. 623)).

Returns — A double which specifies the interest payment for a given period for an investment.

Irr

\[ \text{Irr} \quad \text{static public double Irr(double[] pmt)} \]

Returns the internal rate of return for a schedule of cash flows.

It is found by solving the following:

\[
0 = \sum_{i=1}^{count} \frac{value_i}{(1 + rate)^i}
\]

where \( value_i \) = the \( ith \) cash flow, \( rate \) is the internal rate of return.

pmt  A double array which contains cash flow values which occur at regular intervals.

Returns — A double which specifies the internal rate of return.

Irr

\[ \text{Irr} \quad \text{static public double Irr(double[] pmt, double guess)} \]

Returns the internal rate of return for a schedule of cash flows.

It is found by solving the following:
0 = \sum_{i=1}^{\text{count}} \frac{\text{value}_i}{(1 + \text{rate})^i}

where value\_i = the ith cash flow, rate is the internal rate of return.

**pmt** A double array which contains cash flow values which occur at regular intervals.

**guess** A double value which represents an initial guess at the return value from this function.

Returns — A double which specifies the internal rate of return.

**MIRR**

static public double Mirr(double[] cashFlow, double financeRate, double reinvestRate)

Returns the modified internal rate of return for a schedule of periodic cash flows.

The modified internal rate of return differs from the ordinary internal rate of return in assuming that the cash flows are reinvested at the cost of capital, not at the internal rate of return. It also eliminates the multiple rates of return problem. It is computed using the following:

\[
\left\{ \frac{-(pnpv) (1 + \text{reinvestRate})^n\text{per}}{\text{nnpv} (1 + \text{financeRate})} \right\}^{\frac{1}{\text{nper} - 1}} - 1
\]

where pnpv is calculated from Npv for positive values in values using reinvestRate, and where nnpv is calculated from Npv for negative values in values using financeRate.

**cashFlow** A double array of cash flows.

**financeRate** A double which specifies the interest you pay on the money you borrow.

**reinvestRate** A double which specifies the interest rate you receive on the cash flows.

Returns — A double which specifies the modified internal rate of return.

**Nominal**

static public double Nominal(double effectiveRate, int nper)

Returns the nominal annual interest rate.

The nominal interest rate is the interest rate as stated on the face of a security. It is computed using the following:

\[
\left[ (1 + \text{effectiveRate})^{\frac{1}{\text{nper}}} - 1 \right] \times \text{nper}
\]

**effectiveRate** A double which specifies the effective interest rate.

**nper** A int which specifies the number of compounding periods per year.
Returns — A **double** which specifies the nominal annual interest rate.

**Nper**

```java
static public double Nper(double rate, double pmt, double pv, double fv,
                          Imsl.Finance.Finance.Period period)
```

Returns the number of periods for an investment for which periodic, and constant payments are made and the interest rate is constant.

It can be found by solving the following:

If \( rate = 0 \),

\[
pv + pmt \times nper + fv = 0
\]

If \( rate \neq 0 \),

\[
pv(1 + rate)^{nper} + pmt \left[ 1 + rate \times (period) \right] \left( \frac{(1 + rate)^{nper} - 1}{rate} \right) + fv = 0
\]

- **rate** A **double** which specifies the interest rate.
- **pmt** A **double** which specifies the payment.
- **pv** A **double** which specifies the present value.
- **fv** A **double** which specifies the future value.

**Npv**

```java
static public double Npv(double rate, double[] eqCashFlow)
```

Returns the net present value of a stream of equal periodic cash flows, which are subject to a given discount rate.

It is found by solving the following:

\[
\sum_{i=1}^{count} \frac{value_i}{(1 + rate)^i}
\]

where \( value_i \) = the \( i \)th cash flow.

- **rate** A **double** which specifies the interest rate per period.
- **eqCashFlow** A **double** array of equally-spaced cash flows.

Returns — A **double** which specifies the net present value of the investment.
PeriodicPayment

static public double PeriodicPayment(double rate, int nper, double pv, double fv, Imsl.Finance.Finance.Period period)
Returns the periodic payment for an investment.

It can be found by solving the following:

If \( rate = 0 \),

\[ pv + pmt \times nper + fv = 0 \]

If \( rate \neq 0 \),

\[ pv(1 + rate)^{nper} + pmt[1 + rate(period)] \frac{(1 + rate)^{nper} - 1}{rate} + fv = 0 \]

rate  A double which specifies the interest rate.
nper  A int which specifies the total number of periods.
pv    A double which specifies the present value.
fv    A double which specifies the future value.
period A int which specifies the time in each period when the payment is made (either
        Imsl.Finance.Finance.Period.AtEnd (p. 623) or
        Imsl.Finance.Finance.Period.AtBeginning (p. 623)).

Returns — A double which specifies the interest payment for a given period for an
investment.

Ppmt

static public double Ppmt(double rate, int period, int nper, double pv, double fv, Imsl.Finance.Finance.Period time)
Returns the payment on the principal for a specified period.

It is computed using the following:

\[ payment_i - interest_i \]

where \( payment_i \) is computed from \( pmt \) for the \( i \)-th period, \( interest_i \) is calculated from \( Ipmt \) for the \( i \)-th period.

rate  A double which specifies the interest rate.
period A int which specifies the payment period.
nper  A int which specifies the total number of periods.
pv    A double which specifies the present value.
fv    A double which specifies the future value.
time  The time of a Period when the payment is made (either
Imsl.Finance.Finance.Period.AtEnd (p. 623) or
Imsl.Finance.Finance.Period.AtBeginning (p. 623)).

Returns — A double which specifies the payment on the principal for a given period.

Pv
static public double Pv(double rate, int nper, double pmt, double fv,
Imsl.Finance.Finance.Period time)
Returns the net present value of a stream of equal periodic cash flows, which are subject
to a given discount rate.
It can be found by solving the following:
If rate = 0,
\[ pv + pmt \times nper + fv = 0 \]

If rate ≠ 0,
\[ pv(1 + rate)^{nper} + pmt [1 + rate (\text{period})] \left( \frac{(1 + rate)^{nper} - 1}{rate} \right) + fv = 0 \]

rate  A double which specifies the interest rate per period.
nper  A int which specifies the number of periods.
pmt  A double which specifies the payment made each period.
fv  A double which specifies the annuity’s value after the last payment.
time  The time in a Period when the payment is made (either
Imsl.Finance.Finance.Period.AtEnd (p. 623) or
Imsl.Finance.Finance.Period.AtBeginning (p. 623)).

Returns — A double which specifies the present value of the investment.

Rate
static public double Rate(int nper, double pmt, double pv, double fv,
Imsl.Finance.Finance.Period time)
Returns the interest rate per period of an annuity.
Rate is calculated by iteration and can have zero or more solutions. It can be found by
solving the following:
If rate = 0,
\[ pv + pmt \times nper + fv = 0 \]

If rate ≠ 0,
\[ pv(1 + rate)^{nper} + pmt [1 + rate (\text{period})] \left( \frac{(1 + rate)^{nper} - 1}{rate} \right) + fv = 0 \]
nper  A int which specifies the number of periods.
pmt  A double which specifies the payment made each period.
pv  A double which specifies the present value.
fv  A double which specifies the annuity’s value after the last payment.
time  The time in a Period when the payment is made (either
       Imsl.Finance.Finance.Period.AtEnd (p. 623) or
       Imsl.Finance.Finance.Period.AtBeginning (p. 623)).

Returns — A double which specifies the interest rate per period of an annuity.

Rate
static public double Rate(int nper, double pmt, double pv, double fv,
                          Imsl.Finance.Finance.Period time, double guess)
Returns the interest rate per period of an annuity with an initial guess.
Rate is calculated by iteration and can have zero or more solutions. It can be found by
solving the following:
If rate = 0,
    \[ pv + pmt \times nper + fv = 0 \]
If rate ≠ 0,
    \[ pv(1 + rate)^nper + pmt[1 + rate \times \text{period}] \cdot \frac{(1 + rate)^nper - 1}{rate} + fv = 0 \]

nper  A int which specifies the number of periods.
pmt  A double which specifies the payment made each period.
pv  A double which specifies the present value.
fv  A double which specifies the annuity’s value after the last payment.
time  The time in a Period when the payment is made (either
       Imsl.Finance.Finance.Period.AtEnd (p. 623) or
       Imsl.Finance.Finance.Period.AtBeginning (p. 623)).
guess  A double value which represents an initial guess at the interest rate per period of an annuity.

Returns — A double which specifies the interest rate per period of an annuity.

Sln
static public double Sln(double cost, double salvage, int life)
Returns the depreciation of an asset using the straight line method.
It is computed using the following:
    \[ cost - salvage/life \]
**cost** A double which specifies the initial cost of the asset.
**salvage** A double which specifies the salvage value of the asset.
**life** A int which specifies the number of periods over which the asset is being depreciated.

Returns — A double which specifies the straight line depreciation of an asset for one period.

**Syd**

static public double Syd(double cost, double salvage, int life, int per)

Returns the depreciation of an asset using the sum-of-years digits method.

It is computed using the following:

\[(cost - salvage)(per) \frac{(life + 1)(life)}{2}\]

**cost** A double which specifies the initial cost of the asset.
**salvage** A double which specifies the salvage value of the asset.
**life** A int which specifies the number of periods over which the asset is being depreciated.
**per** A int which specifies the period.

Returns — A double which specifies the sum-of-years digits depreciation of an asset.

**Vdb**

static public double Vdb(double cost, double salvage, int life, int firstPeriod, int lastPeriod, double factor, bool noSL)

Returns the depreciation of an asset for any given period using the variable-declining balance method.

It is computed using the following:

If no_{sl} = 0,

\[\sum_{i=firstPeriod+1}^{lastPeriod} ddb_i\]

If no_{sl} \neq 0,

\[A + \sum_{i=k}^{lastPeriod} \frac{cost - A - salvage}{lastPeriod - k + 1}\]

where ddb\_i is computed from Ddb for the i-th period. k = the first period where straight line depreciation is greater than the depreciation using the double-declining balance method.

\[A = \sum_{i=firstPeriod+1}^{k-1} ddb_i\]
cost  A double which specifies the initial cost of the asset.
salvage  A double which specifies the salvage value of the asset.
life  A int which specifies the number of periods over which the asset is being depreciated.
firstPeriod  A int which specifies the first period for the calculation.
lastPeriod  A int which specifies the last period for the calculation.
factor  A double which specifies the rate at which the balance declines.
noSL  A boolean flag. If true, do not switch to straight-line depreciation even when the depreciation is greater than the declining balance calculation.

Returns — A double which specifies the depreciation of the asset.

Xirr

static public double Xirr(double[] pmt, System.DateTime[] dates)

Returns the internal rate of return for a schedule of cash flows.

It is not necessary that the cash flows be periodic. It can be found by solving the following:

\[ 0 = \sum_{i=1}^{\text{count}} \frac{\text{value}_i}{(1 + \text{rate})^{\frac{d_i - d_1}{365}}} \]

In the equation above, \(d_i\) represents the ith payment date. \(d_1\) represents the 1st payment date. \(\text{value}\) represents the ith cash flow. \(\text{rate}\) is the internal rate of return.

pmt  A double array which contains cash flow values which correspond to a schedule of payments in dates.

dates  A DateTime array which contains a schedule of payment dates.

Returns — A double which specifies the internal rate of return.

Xirr

static public double Xirr(double[] pmt, System.DateTime[] dates, double guess)

Returns the internal rate of return for a schedule of cash flows with a user supplied initial guess.

It is not necessary that the cash flows be periodic. It can be found by solving the following:

\[ 0 = \sum_{i=1}^{\text{count}} \frac{\text{value}_i}{(1 + \text{rate})^{\frac{d_i - d_1}{365}}} \]

In the equation above, \(d_i\) represents the ith payment date. \(d_1\) represents the 1st payment date. \(\text{value}\) represents the ith cash flow. \(\text{rate}\) is the internal rate of return.
**pmt**  A double array which contains cash flow values which correspond to a schedule of payments in dates.

**dates**  A DateTime array which contains a schedule of payment dates.

**guess**  A double value which represents an initial guess at the return value from this function.

Returns — A double which specifies the internal rate of return.

**Xnpv**

static public double Xnpv(double rate, double[] cashFlow, System.DateTime[] dates)

Returns the present value for a schedule of cash flows.

It is not necessary that the cash flows be periodic. It is computed using the following:

\[
\sum_{i=1}^{\text{count}} \frac{\text{value}_i}{(1 + \text{rate})^{(d_i - d_1)/365}}
\]

In the equation above, \(d_i\) represents the \(i\)th payment date, \(d_1\) represents the first payment date, and \(\text{value}_i\) represents the \(i\)th cash flow.

**rate**  A double which specifies the interest rate.

**cashFlow**  A double array containing the cash flows.

**dates**  A DateTime array which contains a schedule of payment dates.

Returns — A double which specifies the present value.

---

**Example: Cumulative Interest Example**

The amount of interest paid in the first year of a 30 year fixed rate mortgage is computed. The amount financed is $200,000 at an interest rate of 7.25% for 30 years.

```csharp
using System;
using Imsl.Finance;

class cumipmtEx1
{
    public static void Main(string[] args)
    {
        double rate = 0.0725 / 12;
        int periods = 12 * 30;
        double pv = 200000;
        int start = 1;
        int end = 12;
        double total = Finance.Cumipmt(rate, periods, pv, start, end, Finance.Period.AtEnd);
        Console.WriteLine("First year interest = "+
total.ToString("C"));
    }
}
```
Example: Cumulative Principal Example

The amount of principal paid in the first year of a 30 year fixed rate mortgage is computed. The amount financed is $200,000 at an interest rate of 7.25% for 30 years.

```csharp
using System;
using Imsl.Finance;

class cumprincEx1
{
    static void Main(string[] args)
    {
        double rate = 0.0725 / 12;
        int periods = 12 * 30;
        double pv = 200000;
        int start = 1;
        int end = 12;
        double total = Imsl.Finance.Cumprinc(rate, periods, pv, start, end, Imsl.Finance.Period.AtEnd);
        Console.WriteLine("First year principal = 
                           " + total.ToString("C"));
    }
}
```

Output

First year principal = ($1,935.71)

Example: Depreciation - Fixed Declining Balance Method

The depreciation of an asset with an initial cost of $2500 and a salvage value of $500 over a period of 3 years is calculated. Here month is 6 since the life of the asset did not begin until the seventh month of the first year.

```csharp
using System;
using Imsl.Finance;

class dbEx1
{
    static void Main(string[] args)
    {
    }
}
```
Example: Depreciation - Double-Declining Balance Method

The depreciation of an asset with an initial cost of $2500 and a salvage value of $500 over a period of 2 years is calculated. A factor of 2 is used (the double-declining balance method).

```csharp
using System;
using Imsl.Finance;

public class ddbEx1
{
    public static void Main(String[] args)
    {
        double cost = 2500;
        double salvage = 500;
        double factor = 2;
        int life = 24;

        for (int period = 1; period <= life; period++)
        {
            double ddb = Finance.Ddb(cost, salvage, life, period, factor);
            Console.Out.WriteLine("For period " + period + " ddb = " + ddb.ToString("C");
        }
    }
}
```

Output

<table>
<thead>
<tr>
<th>Period</th>
<th>Depreciation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$518.75</td>
</tr>
<tr>
<td>2</td>
<td>$822.22</td>
</tr>
<tr>
<td>3</td>
<td>$481.00</td>
</tr>
<tr>
<td>4</td>
<td>$140.69</td>
</tr>
</tbody>
</table>

Finance
Output

For period 1  ddb = $208.33
For period 2  ddb = $190.97
For period 3  ddb = $175.06
For period 4  ddb = $160.47
For period 5  ddb = $147.10
For period 6  ddb = $134.84
For period 7  ddb = $123.60
For period 8  ddb = $113.30
For period 9  ddb = $103.86
For period 10 ddb = $95.21
For period 11 ddb = $87.27
For period 12 ddb = $80.00
For period 13 ddb = $73.33
For period 14 ddb = $67.22
For period 15 ddb = $61.62
For period 16 ddb = $56.48
For period 17 ddb = $51.78
For period 18 ddb = $47.46
For period 19 ddb = $22.09
For period 20 ddb = $0.00
For period 21 ddb = $0.00
For period 22 ddb = $0.00
For period 23 ddb = $0.00
For period 24 ddb = $0.00

Example: Price Conversion - Fractional Dollars

A fractional dollar price, in this case 1 3/8, is converted to a decimal price.

```csharp
using System;
using Imsl.Finance;

public class dollardeEx1
{
    public static void Main(String[] args)
    {
        double fractionalDollar = 1.3;
        int fraction = 8;

        double dollardec = Finance.Dollarde(fractionalDollar, fraction);
        Console.Out.WriteLine("The fractional dollar 1.3 = "+
                              dollardec.ToString("C");
    }
}
```

Output

The fractional dollar 1.3 = $1.38
Example: Price Conversion - Decimal Dollars

A decimal dollar price, in this case $1.38, is converted to a fractional price.

```csharp
using System;
using Imsl.Finance;

public class dollarfrEx1
{
    public static void Main(String[] args)
    {
        double decimalDollar = 1.38;
        int fraction = 8;

        double dollarfr = Finance.Dollarfr(decimalDollar, fraction);
        Console.Out.WriteLine("The decimal dollar $1.38 as a fractional dollar = "+ dollarfr.ToString("0.00");
    }
}
```

Output

The decimal dollar $1.38 as a fractional dollar = 1.30

Example: Effective Rate

In this example the effective interest rate is computed given that the nominal rate is 6.0% and that the interest will be compounded quarterly.

```csharp
using System;
using Imsl.Finance;

public class effectEx1
{
    public static void Main(String[] args)
    {
        double nominalRate = .06;
        int nper = 4;
        double effectiveRate;

        effectiveRate = Finance.Effect(nominalRate, nper);
        Console.Out.WriteLine("The effective rate of the nominal rate, 6.0%, compounded quarterly is "+ effectiveRate.ToString("P");
    }
}
```
The effective rate of the nominal rate, 6.0%, compounded quarterly is 6.14%

Example: Future Value of an Investment

A couple starts setting aside $30,000 a year when they are 45 years old. They expect to earn 5% interest on the money compounded yearly. The future value of the investment is computed for a 20 year period.

```csharp
using System;
using Imsl.Finance;

class fvEx1
{
    public static void Main(String[] args)
    {
        double rate = .05;
        int nper = 20;
        double payment = -30000.00;
        double pv = -30000.00;

        double fv = Finance.Fv(rate, nper, payment, pv,
                                Finance.Period.AtBeginning);
        Console.Out.WriteLine("After 20 years, the value of the " +
                              "investments " + "will be " +
                              fv.ToString("C");
    }
}
```

After 20 years, the value of the investments will be $1,121,176.49

Example: Future Value - Adjustable Rates

An investment of $10,000 is made. The investment will grow at the rate of 5.1% the first year, with the rate increasing by .1% each year thereafter for a total of 5 years. The future value of the investment is computed.

```csharp
using System;
using Imsl.Finance;

class fvscheduleEx1
{
    public static void Main(String[] args)
    {

    
```
double principal = 10000.0;
double[] schedule = new double[]{.050, .051, .052, .053, .054};
double fvschedule;

fvschedule = Finance.Fvschedule(principal, schedule);
Console.Out.WriteLine("After 5 years the $10,000 investment " +
"will have " + "grown to " +
fvschedule.ToString("C");
}

Output

After 5 years the $10,000 investment will have grown to $12,884.77

Example: Interest Payments

The interest due the second year on a $100,000 25 year loan is calculated. The loan is at 8%.

using System;
using Imsl.Finance;

public class ipmtEx1
{
    public static void Main(String[] args)
    {
        double rate = .08;
        int per = 2;
        int nper = 25;
        double pv = 100000.00;
        double fv = 0.0;

        double ipmt = Finance.Ipmt(rate, per, nper, pv, fv,
                                      Finance.Period.AtEnd);
        Console.Out.WriteLine("The interest due the second year on the" +
                              " $100,000 loan is " +
                              ipmt.ToString("C");
    }
}

Output

The interest due the second year on the $100,000 loan is ($7,890.57)
Example: Internal Rate of Return

A farmer buys 10 young cows and a bull for $4500. The first year he does not expect to sell any calves, he just expects to feed them. Thereafter, he expects to be able to sell calves to offset the cost of feed. He expects them to be productive for 9 years, after which time he will liquidate the herd. The internal rate of return is computed after 9 years.

```
using System;
using Imsl.Finance;

class irrEx1
{
    public static void Main(String[] args)
    {
        double[] pmt = new double[]
        {- 4500.0, - 800.0,
         800.0, 800.0,
         600.0, 600.0,
         800.0, 800.0,
         700.0, 3000.0};

        double irr = Finance.Irr(pmt);
        Console.Out.WriteLine("After 9 years, the internal rate of " +
         "return on the cows is " +
         irr.ToString("P"));
    }
}
```

Output

After 9 years, the internal rate of return on the cows is 7.21 %

Example: Modified Internal Rate of Return

A farmer uses a $4500 loan to buy 10 young cows and a bull. The interest rate on the loan is 8%. He expects to reinvest the profits received in any one year in the money market and receive 5.5%. The first year he does not expect to sell any calves, he just expects to feed them. Thereafter, he expects to be able to sell calves to offset the cost of feed. He expects them to be productive for 9 years, after which time he will liquidate the herd. The modified internal rate of return is computed after 9 years.

```
using System;
using Imsl.Finance;

class mirrEx1
{
    public static void Main(String[] args)
    {
```
double[] x = new double[]{ -4500.0, -800.0, 800.0, 800.0, 600.0, 600.0, 800.0, 800.0, 700.0, 3000.0};

double financeRate = .08;
double reinvestRate = .055;
double mirr = Finance.Mirr(x, financeRate, reinvestRate);

Console.Out.WriteLine("After 9 years, the modified internal " +
"rate of return \non the cows is " +
mirr.ToString("P");

Output

After 9 years, the modified internal rate of return
on the cows is 6.66 %

Example: Nominal Rate

In this example the nominal interest rate is computed given that the effective rate is 6.14% and
that the interest has been compounded quarterly.

using System;
using Imsl.Finance;

public class nominalEx1
{
    public static void Main(String[] args)
    {
        double effectiveRate = .0614;
        int nper = 4;

        double nominalRate = Finance.Nominal(effectiveRate, nper);
        Console.Out.WriteLine("The nominal rate of the effective rate," +
            "6.14%, \ncompounded quarterly is " +
            nominalRate.ToString("P");

    }

Output

The nominal rate of the effective rate,6.14%,
compounded quarterly is 6.00 %
Example: Number of Periods for an Investment

Someone obtains a $20,000 loan at 7.25% to buy a car. They want to make $350 a month payments. Here, the number of payments necessary to pay off the loan is computed.

```csharp
using System;
using Imsl.Finance;

public class nperEx1
{
    public static void Main(String[] args)
    {
        double rate = 0.0725 / 12;
        double pmt = -350.0;
        double pv = 20000;
        double fv = 0.0;
        double nperiods;
        nperiods = Finance.Nper(rate, pmt, pv, fv,
                                Finance.Period.AtBeginning);
        Console.Out.WriteLine("Number of payment periods = "+nperiods);
    }
}
```

Output

Number of payment periods = 69.7805113662826

Example: Net Present Value of an Investment

A lady wins a $10 million lottery. The money is to be paid out at the end of each year in $500,000 payments for 20 years. The current treasury bill rate of 6% is used as the discount rate. Here, the net present value of her prize is computed.

```csharp
using System;
using Imsl.Finance;

public class npvEx1
{
    public static void Main(String[] args)
    {
        double rate = 0.06;
        double[] value_Renamed = new double[20];
        for (int i = 0; i < 20; i++)
            value_Renamed[i] = 500000.0;
        double npv = Finance.Npv(rate, value_Renamed);
        Console.Out.WriteLine("The net present value of the $10 "+
                              "million prize is "+npv.ToString("C"));
    }
}
```
The net present value of the $10 million prize is $5,734,960.61

Example: Periodic Payments

The payment due each year on a 25 year, $100,000 loan is calculated. The loan is at 8%.

```csharp
using System;
using Imsl.Finance;

public class pmtEx1
{
    public static void Main(String[] args)
    {
        double rate = .08;
        int nper = 25;
        double pv = 100000.00;
        double fv = 0.0;

        double pmt = Finance.PeriodicPayment(rate, nper, pv, fv,
                                               Finance.Period.AtEnd);
        Console.Out.WriteLine("The payment due each year on the "$100,000 loan is " + pmt.ToString("C");
    }
}
```

Output

The payment due each year on the $100,000 loan is ($9,367.88)

Example: Principal Payments

The payment on the principal the first year on a 25 year, $100,000 loan is calculated. The loan is at 8%.

```csharp
using System;
using Imsl.Finance;

public class ppmtEx1
{
    public static void Main(String[] args)
    {
```

Finance Class • 617
double rate = .08;
int per = 1;
int nper = 25;
double pv = 100000.00;
double fv = 0.0;

double ppmt = Finance.Ppmt(rate, per, nper, pv, fv,
        Finance.Period.AtEnd);
Console.Out.WriteLine("The payment on the principal the first "+
        "year \nof the $100,000 loan is " +
        ppmt.ToString("C");
}
}

Output
The payment on the principal the first year
of the $100,000 loan is ($1,367.88)

Example: Present Value of an Investment

A lady wins a $10 million lottery. The money is to be paid out at the end of each year in
$500,000 payments for 20 years. The current treasury bill rate of 6% is used as the discount
rate. Here, the present value of her prize is computed.

using System;
using Imsl.Finance;

public class pvEx1
{
    public static void Main(String[] args)
    {
        double rate = 0.06;
        double pmt = 500000.0;
        double fv = 0.0;
        int nper = 20;

        double pv = Finance.Pv(rate, nper, pmt, fv,
                Finance.Period.AtEnd);

        Console.Out.WriteLine("The present value of the $10 million " +
                "prize is " + pv.ToString("C");
    }
}

Output
The present value of the $10 million prize is ($5,734,960.61)
Example: Interest Rate

Someone obtains a $20,000 loan to buy a car. They make $350 a month payments for 70 months. Here, the interest rate of the loan is computed.

```csharp
using System;
using Imsl.Finance;

public class rateEx1
{
    public static void Main(String[] args)
    {
        int nper = 70;
        double pmt = -350.0;
        double pv = 20000;
        double fv = 0.0;

        double rate = 12.0 * Imsl.Finance.Rate(nper, pmt, pv, fv,
                                                Imsl.Finance.Period.AtBeginning);
        Console.Out.WriteLine("The computed interest rate on the loan is " + "+" + rate.ToString("P");
    }
}
```

Output

The computed interest rate on the loan is 7.35 %

Example: Depreciation - Straight Line Method

The straight line depreciation for one period of an asset with a life of 24 months, an initial cost of $2500 and a salvage value of $500 is computed.

```csharp
using System;
using Imsl.Finance;

public class slnEx1
{
    public static void Main(String[] args)
    {
        double cost = 2500;
        double salvage = 500;
        int life = 24;

        double sln = Imsl.Finance.Sln(cost, salvage, life);
        Console.Out.WriteLine("The straight line depreciation of the asset for one period is " + "+" + sln.ToString("C");
    }
}
```
Output

The straight line depreciation of the asset for one period is $83.33

Example: Depreciation - Sum-of-years’ Digits

The sum-of-years’ digits depreciation for the 14th year of an asset with a life of 15 years, an initial cost of $25000 and a salvage value of $5000 is computed.

```csharp
using System;
using Imsl.Finance;

public class sydEx1
{
    public static void Main(String[] args)
    {
        double cost = 25000;
        double salvage = 5000;
        int life = 15;
        int per = 14;

        double syd = Finance.Syd(cost, salvage, life, per);
        Console.Out.WriteLine("The depreciation allowance for the 14th" +" year is " + syd.ToString("C");
    }
}
```

Output

The depreciation allowance for the 14th year is $333.33

Example: Depreciation - Variable Declining Balance

The depreciation between the 10th and 15th year of an asset with a life of 15 years, an initial cost of $25000 and a salvage value of $5000 is computed. The variable-declining balance method is used.

```csharp
using System;
using Imsl.Finance;

public class vdbEx1
{
    public static void Main(String[] args)
    {
```
double cost = 25000;
double salvage = 5000;
int life = 15;
int start = 10;
int end = 15;
double factor = 2.0;
bool no_sl = false;

double vdb = Finance.Vdb(cost, salvage, life, start, end,
    factor, no_sl);
Console.Out.WriteLine("The depreciation allowance between the " +
    "10th and 15th year is " +
    vdb.ToString("C");
}

Output

The depreciation allowance between the 10th and 15th year is $976.69

Example: Internal Rate of Return - Variable Schedule

A farmer buys 10 young cows and a bull for $4500. The first year he does not expect to sell any calves, he just expects to feed them. Thereafter, he expects to be able to sell calves to offset the cost of feed. He expects them to be productive for 9 years, after which time he will liquidate the herd. The internal rate of return is computed after 9 years.

using System;
using Imsl.Finance;

public class xirrEx1
{
    public static void Main(String[] args)
    {
        double[] pmt = new double[]{- 4500.0, - 800.0,
            800.0, 800.0,
            600.0, 600.0,
            800.0, 800.0,
            700.0, 3000.0};
        System.DateTime[] dates =
            new System.DateTime[] {DateTime.Parse("1/1/98"),
            DateTime.Parse("10/1/98"),
            DateTime.Parse("5/5/99"),
            DateTime.Parse("5/5/00"),
            DateTime.Parse("6/1/01"),
            DateTime.Parse("7/1/02"),
            DateTime.Parse("8/30/03"),
            DateTime.Parse("9/15/04"),
            DateTime.Parse("10/15/05"),
            DateTime.Parse("11/1/06");

Finance

Finance Class • 621
double xirr = Finance.Xirr(pmt, dates);

Console.Out.WriteLine("After approximately 9 years, the " +
  "internal rate of return \n" +
  "on the cows is " + xirr.ToString("P");
}
}

Output

After approximately 9 years, the internal rate of return
on the cows is 7.69 %

Example: Present Value of a Schedule of Cash Flows

In this example, the present value of 3 payments, $1,000, $2,000, and $1,000, with an interest
rate of 5% made on January 3, 1997, January 3, 1999, and January 3, 2000 is computed.

using System;
using Imsl.Finance;

public class xnpvEx1
{
    public static void Main(String[] args)
    {
        double rate = 0.05;
        double[] value_Renamed = new double[]{1000.0, 2000.0, 1000.0};
        System.DateTime[] dates =
            new System.DateTime[]{DateTime.Parse("1/3/1997"),
                                   DateTime.Parse("1/3/1999"),
                                   DateTime.Parse("1/3/2000")};

        double pv = Finance.Xnpv(rate, value_Renamed, dates);
        Console.Out.WriteLine("The present value of the schedule of " +
                              "cash flows is " + pv.ToString("C");
    }
}

Output

The present value of the schedule of cash flows is $3,677.90
Finance.Period Enumeration

Used to indicate that payment is made at the beginning or end of each period.

public enumeration Imsl.Finance.Finance.Period

Fields

AtBeginning
  public Imsl.Finance.Finance.Period AtBeginning
  Indicates payment is made at the beginning of each period.

AtEnd
  public Imsl.Finance.Finance.Period AtEnd
  Indicates payment is made at the end of each period.

IBasisPart Interface

Component of DayCountBasis.

public interface Imsl.Finance.IBasisPart

Methods

DaysBetween
  abstract public int DaysBetween(System.DateTime date1, System.DateTime date2)
  Returns the number of days from date1 to date2.

  date1  A DateTime object containing the initial date.
  date2  A DateTime object containing the final date.

  Returns — A int which specifies the number of days from date1 to date2.

DaysInPeriod
  abstract public double DaysInPeriod(System.DateTime finalDate, Imsl.Finance.Bond.Frequency frequency)
  Returns the number of days in a coupon period.

  finalDate  A DateTime object containing the final date of the coupon period.
The Frequency specifying the number of coupon periods per year. This is typically 1, 2 or 4.

Returns — A int containing the number of days in the coupon period.

GetDaysInYear

abstract public int GetDaysInYear(System.DateTime settlement, System.DateTime maturity)
Returns the number of days in the year.

settlement  A DateTime object containing the settlement date.
maturity   A DateTime object containing the maturity date.

Returns — A int which specifies the number of days in the year.

Description

The day count basis consists of a month basis and a yearly basis. Each of these components implements this interface.

See Also

Imsl.Finance.DayCountBasis (p. 657)

Bond Class

Collection of bond functions.

public class Imsl.Finance.Bond

Methods

Accrint

static public double Accrint(System.DateTime issue, System.DateTime firstCoupon, System.DateTime settlement, double rate, double par, Imsl.Finance.Bond.Frequency frequency, Imsl.Finance.DayCountBasis basis)
Returns the interest which has accrued on a security that pays interest periodically.

In the equation below, \( A_i \) represents the number of days which have accrued for the \( i \)th quasi-coupon period within the odd Frequency. (The quasi-coupon periods are periods obtained by extending the series of equal payment periods to before or after the actual payment periods.) \( NC \) represents the number of quasi-coupon periods within the odd
period, rounded to the next highest integer. (The odd period is a period between
payments that differs from the usual equally spaced periods at which payments are
made.) $NL_i$ represents the length of the normal $i$th quasi-coupon period within the odd
Frequency. $NL_i$ is expressed in days. Accrint solves the following:

$$par \left( \frac{rate}{frequency} \sum_{i=1}^{NC} \frac{A_i}{NL_i} \right)$$

**issue**  The DateTime issue date of the security.

**firstCoupon**  The DateTime date of the security’s first interest date.

**settlement**   The DateTime settlement date of the security.

**rate**  A double which specifies the security’s annual coupon rate.

**par**  A double which specifies the security’s par value.

**frequency**  A int which specifies the number of coupon payments per year (1 for
annual, 2 for semiannual, 4 for quarterly).

**basis**  A DayCountBasis object which contains the type of day count basis to use.

Returns — A double which specifies the accrued interest.

**Accrintm**

```csharp
static public double Accrintm(System.DateTime issue, System.DateTime maturity, double rate, double par, Imsl.Finance.DayCountBasis basis)
```

Returns the interest which has accrued on a security that pays interest at maturity.

$$= par \times rate \times \frac{A}{D}$$

In the above equation, $A$ represents the number of days starting at issue date to maturity
date and $D$ represents the annual basis.

**issue**  Ahe DateTime issue date of the security.

**maturity**   The DateTime date of the security’s maturity.

**rate**  A double which specifies the security’s annual coupon rate.

**par**  A double which specifies the security’s par value.

**basis**  A DayCountBasis object which contains the type of day count basis to use.

Returns — A double which specifies the accrued interest.

**Amordegrc**

```csharp
static public double Amordegrc(double cost, System.DateTime issue, System.DateTime firstPeriod, double salvage, int period, double rate, Imsl.Finance.DayCountBasis basis)
```

Returns the depreciation for each accounting Frequency.

This function is similar to Amorlinc. However, in this function a depreciation coefficient
based on the asset life is applied during the evaluation of the function.
cost  A double which specifies the cost of the asset.
issue  The DateTime issue date of the asset.
firstPeriod  The DateTime date of the end of the first period.
salvage  A double which specifies the asset’s salvage value at the end of the life of the asset.
period  A int which specifies the period.
rate  A double which specifies the rate of depreciation.
basis  A DayCountBasis object which contains the type of day count basis to use.

Returns — A double which specifies the depreciation.

Amorlinc
static public double Amorlinc(double cost, System.DateTime issue, System.DateTime firstPeriod, double salvage, int period, double rate, Imsl.Finance.DayCountBasis basis)
Returns the depreciation for each accounting Frequency.

This function is similar to Amordegrc, except that Amordegrc has a depreciation coefficient that is applied during the evaluation that is based on the asset life.

cost  A double which specifies the cost of the asset.
issue  The DateTime issue date of the asset.
firstPeriod  The DateTime date of the end of the first period.
salvage  A double which specifies the asset’s salvage value at the end of the life of the asset.
period  A int which specifies the period.
rate  A double which specifies the rate of depreciation.
basis  A DayCountBasis object which contains the type of day count basis to use.

Returns — A double which specifies the depreciation.

Convexity
static public double Convexity(System.DateTime settlement, System.DateTime maturity, double coupon, double yield, Imsl.Finance.Bond.Frequency frequency, Imsl.Finance.DayCountBasis basis)
Returns the convexity for a security.

Convexity is the sensitivity of the duration of a security to changes in yield. It is computed using the following:

\[\frac{1}{(q \times \text{frequency})^2} \left\{ \sum_{t=1}^{n} t(t+1) \left( \frac{\text{coupon}}{\text{frequency}} \right) q^{-t} + n(n+1) q^{-n} \right\} \]

\[\left( \sum_{t=1}^{n} \left( \frac{\text{coupon}}{\text{frequency}} \right) q^{-t} + q^{-n} \right)\]

where n is calculated from Coupnum, and \( q = 1 + \frac{\text{yield}}{\text{frequency}} \).
settlement  The DateTime settlement date of the security.
maturity  The DateTime maturity date of the security.
coupon   A double which specifies the security’s annual coupon rate.
yield    A double which specifies the security’s annual yield.
frequency A int which specifies the number of coupon payments per year (1 for annual, 2 for semiannual, 4 for quarterly).
basis    A DayCountBasis object which contains the type of day count basis to use.

Returns — A double which specifies the convexity for a security.

Coupdaybs
static public int Coupdaybs(System.DateTime settlement, System.DateTime maturity, Imsl.Finance.Bond.Frequency frequency, Imsl.Finance.DayCountBasis basis)
Returns the number of days starting with the beginning of the coupon period and ending with the settlement date.

For a good discussion on day count basis, see SIA Standard Securities Calculation Methods 1993, vol. 1, pages 17-35.

settlement  The DateTime settlement date of the security.
maturity  The DateTime maturity date of the security.
frequency A int which specifies the number of coupon payments per year.
basis    A DayCountBasis object which contains the type of day count basis to use.

Returns — A int which specifies the number of days from the beginning of the coupon period to the settlement date.

Coupdays
static public double Coupdays(System.DateTime settlement, System.DateTime maturity, Imsl.Finance.Bond.Frequency frequency, Imsl.Finance.DayCountBasis basis)
Returns the number of days in the coupon period containing the settlement date.

For a good discussion on day count basis, see SIA Standard Securities Calculation Methods 1993, vol. 1, pages 17-35.

settlement  The DateTime settlement date of the security.
maturity  The DateTime maturity date of the security.
frequency A int which specifies the number of coupon payments per year.
basis    A DayCountBasis object which contains the type of day count basis to use.

Returns — A int which specifies the number of days in the coupon period that contains the settlement date.
Coupdaysnc

```
static public int Coupdaysnc(System.DateTime settlement, System.DateTime maturity, Imsl.Finance.Bond.Frequency frequency, Imsl.Finance.DayCountBasis basis)
```

Returns the number of days starting with the settlement date and ending with the next coupon date.

For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

- **settlement**  The `DateTime` settlement date of the security.
- **maturity**  The `DateTime` maturity date of the security.
- **frequency**  A `int` which specifies the number of coupon payments per year.
- **basis**  A `DayCountBasis` object which contains the type of day count basis to use.

Returns — A `int` which specifies the number of days from the settlement date to the next coupon date.

Coupncd

```
```

Returns the first coupon date which follows the settlement date.

For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

- **settlement**  The `DateTime` settlement date of the security.
- **maturity**  The `DateTime` maturity date of the security.
- **frequency**  A `int` which specifies the number of coupon payments per year.
- **basis**  A `DayCountBasis` object which contains the type of day count basis to use.

Returns — A `int` which specifies the next coupon date after the settlement date.

Coupnum

```
static public int Coupnum(System.DateTime settlement, System.DateTime maturity, Imsl.Finance.Bond.Frequency frequency, Imsl.Finance.DayCountBasis basis)
```

Returns the number of coupons payable between the settlement date and the maturity date.

For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

- **settlement**  The `DateTime` settlement date of the security.
- **maturity**  The `DateTime` maturity date of the security.
- **frequency**  A `int` which specifies the number of coupon payments per year.
- **basis**  A `DayCountBasis` object which contains the type of day count basis to use.
Returns — A int which specifies the number of coupons payable between the settlement date and maturity date.

Couppcd

```csharp
static public System.DateTime Couppcd(System.DateTime settlement,
System.DateTime maturity, Imsl.Finance.Bond.Frequency frequency,
Imsl.Finance.DayCountBasis basis)
```

Returns the coupon date which immediately precedes the settlement date.

For a good discussion on day count basis, see SIA Standard Securities Calculation Methods 1993, vol. 1, pages 17-35.

**settlement**  The DateTime settlement date of the security.

**maturity**  The DateTime maturity date of the security.

**frequency**  A int which specifies the number of coupon payments per year.

**basis**  A DayCountBasis object which contains the type of day count basis to use.

Returns — A int which specifies the previous coupon date before the settlement date.

Disc

```csharp
static public double Disc(System.DateTime settlement, System.DateTime maturity,
double price, double redemption, Imsl.Finance.DayCountBasis basis)
```

Returns the implied interest rate of a discount bond.

The discount rate is the interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments. It is computed using the following:

\[
\frac{\text{redemption} - \text{price}}{\text{price}} \times \frac{B}{DSM}
\]

In the equation above, \( B \) represents the number of days in a year based on the annual basis and \( DSM \) represents the number of days starting with the settlement date and ending with the maturity date.

**settlement**  The DateTime settlement date of the security.

**maturity**  The DateTime maturity date of the security.

**price**  A double which specifies the security’s price per $100 face value.

**redemption**  A double which the security’s redemption value per $100 face value.

**basis**  A DayCountBasis object which contains the type of day count basis to use.

Returns — A double which specifies the discount rate for a security.

Duration

```csharp
static public double Duration(System.DateTime settlement, System.DateTime maturity,
double coupon, double yield, Imsl.Finance.Bond.Frequency frequency,
Imsl.Finance.DayCountBasis basis)
```

Finance  Bond Class • 629
Returns the Macauley’s duration of a security where the security has periodic interest payments.

The Macauley’s duration is the weighted-average time to the payments, where the weights are the present value of the payments. It is computed using the following:

$$D = \frac{\sum_{k=1}^{N} \left( \frac{100 \times \text{coupon}}{\text{freq} \times (1 + \frac{\text{yield}}{\text{freq}})^{(k-1+\frac{\text{DSC}}{\text{E}})}} \times (k - 1 + \frac{\text{DSC}}{\text{E}}) \right)}{\left( \frac{100}{(1 + \frac{\text{yield}}{\text{freq}})^{N-1+\frac{\text{DSC}}{\text{E}}}} \times \sum_{k=1}^{N} \left( \frac{100 \times \text{coupon}}{\text{freq} \times (1 + \frac{\text{yield}}{\text{freq}})^{(k-1+\frac{\text{DSC}}{\text{E}})}} \right) \right) \times \frac{1}{\text{freq}}} \times \frac{100}{(1 + \frac{\text{yield}}{\text{freq}})^{(N-1+\frac{\text{DSC}}{\text{E}})}}$$

In the equation above, $DSC$ represents the number of days starting with the settlement date and ending with the next coupon date. $E$ represents the number of days within the coupon Frequency. $N$ represents the number of coupons payable from the settlement date to the maturity date. $freq$ represents the frequency of the coupon payments annually.

- **settlement** The `DateTime` settlement date of the security.
- **maturity** The `DateTime` maturity date of the security.
- **coupon** A `double` which specifies the security’s annual coupon rate.
- **yield** A `double` which specifies the security’s annual yield.
- **frequency** A `int` which specifies the number of coupon payments per year (1 for annual, 2 for semiannual, 4 for quarterly).
- **basis** A `DayCountBasis` object which contains the type of day count basis to use.

Returns — A `double` which specifies the annual duration of a security with periodic interest payments.

Intrate

static public double Intrate(System.DateTime settlement, System.DateTime maturity, double investment, double redemption, Imsl.Finance.DayCountBasis basis)

Returns the interest rate of a fully invested security.

It is computed using the following:

$$\frac{\text{redemption} - \text{investment}}{\text{investment}} \times \frac{B}{\text{DSM}}$$

In the equation above, $B$ represents the number of days in a year based on the annual basis, and $DSM$ represents the number of days in the period starting with the settlement date and ending with the maturity date.

- **settlement** The `DateTime` settlement date of the security.
- **maturity** The `DateTime` maturity date of the security.
investment  A double which specifies the amount invested.
redemption  A double which specifies the amount to be received at maturity.
basis  A DayCountBasis object which contains the type of day count basis to use.

Returns — A double which specifies the interest rate for a fully invested security.

Mduration

static public double Mduration(System.DateTime settlement, System.DateTime maturity, double coupon, double yield, Imsl.Finance.Bond.Frequency frequency, Imsl.Finance.DayCountBasis basis)

Returns the modified Macauley duration for a security with an assumed par value of $100.

It is computed using the following:

$$
duration = \frac{N \sum_{k=1}^{N} \frac{100 \times \text{rate}}{\text{frequency}} (1 + \frac{\text{yield}}{\text{frequency}})^{k-1+\frac{\text{DSC}}{E}} - \left(100 \times \frac{\text{rate}}{\text{frequency}} \times \frac{\text{redemption}}{E}\right)}{1 + \frac{\text{yield}}{\text{frequency}}}
$$

where duration is calculated from Mduration.

settlement  The DateTime settlement date of the security.
maturity  The DateTime maturity date of the security.
coupon  A double which specifies the security’s annual coupon rate.
yield  A double which specifies the security’s annual yield.
frequency  A int which specifies the number of coupon payments per year (1 for annual, 2 for semiannual, 4 for quarterly).
basis  A DayCountBasis object which contains the type of day count basis to use.

Returns — A double which specifies the modified Macauley duration for a security with an assumed par value of $100.

Price

static public double Price(System.DateTime settlement, System.DateTime maturity, double rate, double yield, double redemption, Imsl.Finance.Bond.Frequency frequency, Imsl.Finance.DayCountBasis basis)

Returns the price, per $100 face value, of a security that pays periodic interest.

It is computed using the following:

$$
\frac{\text{redemption}}{\left(1 + \frac{\text{yield}}{\text{frequency}}\right)^{N-1+\frac{\text{DSC}}{E}}} + \sum_{k=1}^{N} \frac{100 \times \text{rate}}{\text{frequency}} \left(1 + \frac{\text{yield}}{\text{frequency}}\right)^{k-1+\frac{\text{DSC}}{E}} - \left(100 \times \frac{\text{rate}}{\text{frequency}} \times \frac{\text{redemption}}{E}\right)
$$

In the above equation, DSC represents the number of days in the period starting with the settlement date and ending with the next coupon date. E represents the number of days.
within the coupon Frequency. $N$ represents the number of coupons payable in the
timeframe from the settlement date to the redemption date. $A$ represents the number of
days in the timeframe starting with the beginning of coupon period and ending with the
settlement date.

**settlement** The DateTime settlement date of the security.

**maturity** The DateTime maturity date of the security.

**rate** A double which specifies the security’s annual coupon rate.

**yield** A double which specifies the security’s annual yield.

**redemption** A double which specifies the security’s redemption value per $100 face
value.

**frequency** A int which specifies the number of coupon payments per year (1 for
annual, 2 for semiannual, 4 for quarterly).

**basis** A DayCountBasis object which contains the type of day count basis to use.

Returns — A double which specifies the price per $100 face value of a security that pays
periodic interest.

**Pricedisc**

```csharp
static public double Pricedisc(System.DateTime settlement, System.DateTime
maturity, double rate, double redemption, Imsl.Finance.DayCountBasis basis)
```

Returns the price of a discount bond given the discount rate.

It is computed using the following:

\[
redemption - rate \times redemption \times \frac{DSM}{B}
\]

In the equation above, $DSM$ represents the number of days starting at the settlement
date and ending with the maturity date. $B$ represents the number of days in a year based
on the annual basis.

**settlement** The DateTime settlement date of the security.

**maturity** The DateTime maturity date of the security.

**rate** A double which specifies the security’s discount rate.

**redemption** A double which specifies the security’s redemption value per $100 face
value.

**basis** A DayCountBasis object which contains the type of day count basis to use.

Returns — A double which specifies the price per $100 face value of a discounted
security.

**Pricemat**

```csharp
static public double Pricemat(System.DateTime settlement, System.DateTime
maturity, System.DateTime issue, double rate, double yield,
Imsl.Finance.DayCountBasis basis)
```

632 • Bond Class IMLC# Numerical Library
Returns the price, per $100 face value, of a discount bond.

It is computed using the following:

\[
\frac{100 + \left( \frac{\text{DIM} \times \text{rate} \times 100}{B} \right)}{1 + \left( \frac{\text{DSM}}{B} \times \text{yield} \right)} - \frac{\text{A} \times \text{rate} \times 100}{B}
\]

In the equation above, \( B \) represents the number of days in a year based on the annual basis. DSM represents the number of days in the period starting with the settlement date and ending with the maturity date. DIM represents the number of days in the period starting with the issue date and ending with the maturity date. \( A \) represents the number of days in the period starting with the issue date and ending with the settlement date.

**settlement** The DateTime settlement date of the security.

**maturity** The DateTime maturity date of the security.

**issue** The DateTime issue date of the security.

**rate** A double which specifies the security’s interest rate at issue date.

**yield** A double which specifies the security’s annual yield.

**basis** A DayCountBasis object which contains the type of day count basis to use.

Returns — A double which specifies the price per $100 face value of a security that pays interest at maturity.

Priceyield

static public double Priceyield(System.DateTime settlement, System.DateTime maturity, double yield, double redemption, Imsl.Finance.DayCountBasis basis)

Returns the price of a discount bond given the yield.

It is computed using the following:

\[
\frac{\text{redemption}}{1 + \left( \frac{\text{DSM}}{B} \right) \times \text{yield}}
\]

In the equation above, \( DSM \) represents the number of days starting at the settlement date and ending with the maturity date. \( B \) represents the number of days in a year based on the annual basis.

**settlement** The DateTime settlement date of the security.

**maturity** The DateTime maturity date of the security.

**yield** A double which specifies the security’s yield.

**redemption** A double which specifies the security’s redemption value per $100 face value.

**basis** A DayCountBasis object which contains the type of day count basis to use.
Returns — A double which specifies the price per $100 face value of a discounted security.

Received

static public double Received(System.DateTime settlement, System.DateTime maturity, double investment, double rate, Imsl.Finance.DayCountBasis basis)

Returns the amount one receives when a fully invested security reaches the maturity date.

It is computed using the following:

\[
\frac{investment}{1 - (rate \times \frac{DIM}{B})}
\]

In the equation above, \( B \) represents the number of days in a year based on the annual basis, and \( DIM \) represents the number of days in the period starting with the issue date and ending with the maturity date.

**settlement**  The DateTime settlement date of the security.

**maturity**  The DateTime maturity date of the security.

**investment**  A double which specifies the amount invested in the security.

**rate**  A double which specifies the security’s rate at issue date.

**basis**  A DayCountBasis object which contains the type of day count basis to use.

Returns — A double which specifies the amount received at maturity for a fully invested security.

Tbilleq

static public double Tbilleq(System.DateTime settlement, System.DateTime maturity, double rate)

Returns the bond-equivalent yield of a Treasury bill.

It is computed using the following:

If \( DSM \leq 182 \)

\[
\frac{365 \times rate}{360 - rate \times DSM}
\]

otherwise,

\[
-\frac{DSM}{365} + \sqrt{\left(\frac{DSM}{365}\right)^2 - \left(2 \times \frac{DSM}{360} - 1\right) \times \frac{rate \times DSM}{rate \times DSM - 360}} - 0.5
\]

In the above equation, \( DSM \) represents the number of days starting at settlement date to maturity date.

**settlement**  The DateTime settlement date of the Treasury bill.
maturity  The DateTime maturity date of the Treasury bill. The maturity cannot be
more than a year after the settlement.
rate    A double which specifies the Treasury bill’s discount rate at issue date. The
discount rate is an annualized rate of return based on the par value of the bills. The
discount rate is calculated on a 360-day basis (twelve 30-day months).

Returns — A double which specifies the bond-equivalent yield for the Treasury bill. This
is an annualized rate based on the purchase price of the bills and reflects the actual yield
to maturity.

Tbillprice
static public double Tbillprice(System.DateTime settlement, System.DateTime
maturity, double rate)
Returns the price, per $100 face value, of a Treasury bill.
It is computed using the following:

\[
100 \left( 1 - \frac{rate \times DSM}{360} \right)
\]

In the equation above, DSM represents the number of days in the period starting with
the settlement date and ending with the maturity date (any maturity date that is more
than one calendar year after the settlement date is excluded).

settlement  The DateTime settlement date of the Treasury bill.
maturity  The DateTime maturity date of the Treasury bill. The maturity cannot be
more than a year after the settlement.
rate    A double which specifies the Treasury bill’s discount rate at issue date. The
discount rate is an annualized rate of return based on the par value of the bills. The
discount rate is calculated on a 360-day basis (twelve 30-day months).

Returns — A double which specifies the price per $100 face value for the Treasury bill.

Tbillyield
static public double Tbillyield(System.DateTime settlement, System.DateTime
maturity, double price)
Returns the yield of a Treasury bill.
It is computed using the following:

\[
\frac{100 - price}{price} \times \frac{360}{DSM}
\]

In the equation above, DSM represents the number of days in the period starting with
the settlement date and ending with the maturity date (any maturity date that is more
than one calendar year after the settlement date is excluded).

settlement  The DateTime settlement date of the Treasury bill.
**maturity**  The `DateTime` maturity date of the Treasury bill. The maturity cannot be more than a year after the settlement.

**price**  A `double` which specifies the Treasury bill's price per $100 face value. Returns — A `double` which specifies the yield for the Treasury bill. This is an annualized rate based on the purchase price of the bills and reflects the actual yield to maturity.

**Yearfrac**

```csharp
static public double Yearfrac(System.DateTime startDate, System.DateTime endDate, Imsl.Finance.DayCountBasis basis)
```

Returns the fraction of a year represented by the number of whole days between two dates.

It is computed using the following:

\[ \frac{A}{D} \]

where \( A \) equals the number of days from `startDate` to `endDate`, \( D \) equals annual basis.

**startDate**  The `DateTime` start date of the security.

**endDate**  The `DateTime` end date of the security.

**basis**  A `DayCountBasis` object which contains the type of day count basis to use.

Returns — A `double` which specifies the annual yield of a security that pays interest at maturity.

**Yield**

```csharp
static public double Yield(System.DateTime settlement, System.DateTime maturity, double rate, double price, double redemption,
```

Returns the yield of a security that pays periodic interest.

If there is one coupon period use the following:

\[
\left( \frac{\text{redemption}}{100} + \frac{\text{rate}}{\text{frequency}} \right) - \left( \frac{\text{price}}{100} + \left( \frac{A}{E} \times \frac{\text{rate}}{\text{frequency}} \right) \right) \times \frac{\text{frequency} \times E}{\text{DSR}}
\]

In the equation above, \( DSR \) represents the number of days in the period starting with the settlement date and ending with the redemption date. \( E \) represents the number of days within the coupon Frequency. \( A \) represents the number of days in the period starting with the beginning of coupon period and ending with the settlement date.

If there is more than one coupon period use the following:

\[
\text{price} - \frac{\text{redemption}}{1 + \frac{\text{rate}}{\text{frequency}}} - \left( \sum_{k=1}^{N} \frac{100 \times \frac{\text{rate}}{\text{frequency}}}{\left( \frac{E-1+\frac{\text{rate}}{\text{frequency}}}{k-1+\frac{\text{rate}}{\text{frequency}}} \right)} \right) + 100 \times \frac{\text{rate}}{\text{frequency}} \times \frac{A}{E} = 0
\]
In the equation above, \( DSC \) represents the number of days in the period from the settlement to the next coupon date. \( E \) represents the number of days within the coupon frequency. \( N \) represents the number of coupons payable in the period starting with the settlement date and ending with the redemption date. \( A \) represents the number of days in the period starting with the beginning of the coupon period and ending with the settlement date.

**settlement**  The `DateTime` settlement date of the security.

**maturity**  The `DateTime` maturity date of the security.

**rate**  A double which specifies the security’s annual coupon rate.

**price**  A double which specifies the security’s price per $100 face value.

**redemption**  A double which specifies the security’s redemption value per $100 face value.

**frequency**  A int which specifies the number of coupon payments per year (1 for annual, 2 for semiannual, 4 for quarterly).

**basis**  A `DayCountBasis` object which contains the type of day count basis to use.

Returns — A double which specifies the yield of a security that pays periodic interest.

YieldDisc

```
static public double YieldDisc(System.DateTime settlement, System.DateTime maturity, double price, double redemption, Imsl.Finance.DayCountBasis basis)
```

Returns the annual yield of a discount bond.

It is computed using the following:

\[
\frac{\text{redemption} - \text{price}}{\text{price}} \times \frac{B}{DSM}
\]

In the equation above, \( B \) represents the number of days in a year based on the annual basis, and \( DSM \) represents the number of days starting with the settlement date and ending with the maturity date.

**settlement**  The `DateTime` settlement date of the security.

**maturity**  The `DateTime` maturity date of the security.

**price**  A double which specifies the security’s price per $100 face value.

**redemption**  A double which specifies the security’s redemption value per $100 face value.

**basis**  A `DayCountBasis` object which contains the type of day count basis to use.

Returns — A double which specifies the annual yield for a discounted security.
Yieldmat

    static public double Yieldmat(System.DateTime settlement, System.DateTime maturity, System.DateTime issue, double rate, double price, Imsl.Finance.DayCountBasis basis)
    Returns the annual yield of a security that pays interest at maturity.

    It is computed using the following:

    \[
    \frac{
    1 + \left( \frac{DIM}{B} \times rate \right)
    - \left( \frac{price}{100} + \left( \frac{A}{B} \times rate \right) \right)
    \frac{price}{100} + \left( \frac{A}{B} \times rate \right)
    \times DSM}
    \]

    In the equation above, \( DIM \) represents the number of days in the period starting with the issue date and ending with the maturity date. \( DSM \) represents the number of days in the period starting with the settlement date and ending with the maturity date. \( A \) represents the number of days in the period starting with the issue date and ending with the settlement date. \( B \) represents the number of days in a year based on the annual basis.

    settlement  The DateTime settlement date of the security.
    maturity   The DateTime maturity date of the security.
    issue      The DateTime issue date of the security.
    rate       A double which specifies the security’s interest rate at date of issue.
    price      A double which specifies the security’s price per $100 face value.
    basis      A DayCountBasis object which contains the type of day count basis to use.

    Returns — A double which specifies the annual yield of a security that pays interest at maturity.

Description

Definitions

rate is an annualized rate of return based on the par value of the bills.

yield is an annualized rate based on the purchase price and reflects the actual yield to maturity.
coupons are interest payments on a bond.

redemption is the amount a bond pays at maturity.

frequency is the number of times a year that a bond makes interest payments.

basis is the method used to calculate dates. For example, sometimes computations are done assuming 360 days in a year.

issue is the day a bond is first sold.

settlement is the day a purchaser acquires a bond.
maturity is the day a bond’s principal is repaid.
Discount Bonds

Discount bonds, also called zero-coupon bonds, do not pay interest during the life of the security, instead they sell at a discount to their value at maturity. The discount bond methods all have settlement, maturity, basis and redemption as arguments. In the following list these common arguments are ommitted.

- price = Pricedisc(rate) (p. 632)
- price = Priceyield(yield) (p. 633)
- price = Pricemat(issue, rate, yield) (p. 632)
- rate = Disc(price) (p. 629)
- yield = Yielddisc(price) (p. 637)

A related method is Accrintm (p. 625), which returns the interest that has accumulated on the discount bond.

Treasury Bills

US Treasury bills are a special case of discount bonds. The basis is fixed for treasury bills and the redemption value is assumed to be $100. So these functions have only settlement and maturity as common arguments.

- price = Tbillprice(rate) (p. 635)
- yield = Tbillyield(Price) (p. 635)
- yield = Tbillev(rate) (p. 634)

Interest Paying Bonds

Most bonds pay interest periodically. The interest paying bond methods all have settlement, maturity, basis and frequency as arguments. Again supressing the common arguments,

- price = Price(rate, yield, redemption) (p. 631)
- yield = Yield(rate, Price, redemption) (p. 636)
- redemption = Received(Price, rate) (p. 634)

A related method is Accrint (p. 624), which returns the interest that has accumulated at settlement from the previous coupon date.
**Coupon days**

In this diagram, the settlement date is shown as a hollow circle and the adjacent coupon dates are shown as filled circles.

- **Coupncd** (p. 629) is the coupon date immediately prior to the settlement date.
- **Couppcd** (p. 628) is the coupon date immediately after the settlement date.
- **Coupdaysbs** (p. 627) is the number of days from the immediately prior coupon date to the settlement date.
- **Coupdaysnc** (p. 627) is the number of days from the settlement date to the next Coupon date.
- **Coupdays** (p. 627) is the number of days between these two coupon dates.

A related method is **Coupnum** (p. 628), which returns the number of coupons payable between settlement and maturity.
Another related method is Yearfrac (p. 636), which returns the fraction of the year between two days.

**Duration**

Duration is used to measure the sensitivity of a bond to changes in interest rates. Convexity is a measure of the sensitivity of duration.

- Duration (p. 629)
- DayCountBasis modified duration (p. 631)
- Convexity (p. 626)

**Example: Accrued Interest - Periodic Payments**

In this example, the accrued interest is calculated for a bond which pays interest semiannually. The day count basis used is 30/360.

```csharp
using System;
using Imsl.Finance;

public class accrintEx1
{
    public static void Main(String[] args)
    {
        DateTime issue = DateTime.Parse("10/1/91");
        DateTime firstCoupon = DateTime.Parse("3/31/92");
        DateTime settlement = DateTime.Parse("11/3/91");
        double rate = .06;
        double par = 1000.0;
        Bond.Frequency freq = Bond.Frequency.SemiAnnual;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double accrint = Bond.Accrint(issue, firstCoupon, settlement, rate, par, freq, dcb);
        Console.Out.WriteLine("The accrued interest is "+ accrint);
    }
}
```

**Output**

The accrued interest is 5.33333333333333

**Example: Accrued Interest - Payment at Maturity**

In this example, the accrued interest is calculated for a bond which pays at maturity. The day count basis used is 30/360.
using System;
using Imsl.Finance;

class accrintmEx1
{
    public static void Main(string[] args)
    {
        DateTime issue = DateTime.Parse("10/1/91");
        DateTime settlement = DateTime.Parse("11/3/91");
        double rate = 0.06;
        double par = 1000.0;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double accrintm = Bond.Accrintm(issue, settlement, rate, par, dcb);
        Console.WriteLine("The accrued interest is "+ accrintm);
    }
}

Output
The accrued interest is 5.33333333333333

Example: Depreciation - French Accounting System

In this example, the depreciation for the second accounting period is calculated for an asset.

class amordegrcEx1
{
    public static void Main(string[] args)
    {
        double cost = 2400.0;
        DateTime issue = DateTime.Parse("11/1/92");
        DateTime firstPeriod = DateTime.Parse("11/30/93");
        double salvage = 300.0;
        int period = 2;
        double rate = 0.15;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double amordegrc = Bond.Amordegrc(cost, issue, firstPeriod, salvage, period, rate, dcb);
        Console.WriteLine("The depreciation for the second accounting "+ "period is " + amordegrc);
    }
}

Output
The depreciation for the second accounting period is 334
Example: Depreciation - French Accounting System

In this example, the depreciation for the second accounting period is calculated for an asset.

```csharp
using System;
using Imsl.Finance;

public class amorlincEx1
{
    public static void Main(String[] args)
    {
        double cost = 2400.0;
        DateTime issue = DateTime.Parse("11/1/92");
        DateTime firstPeriod = DateTime.Parse("11/30/93");
        double salvage = 300.0;
        int period = 2;
        double rate = .15;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double amorlinc = Bond.Amorlinc(cost, issue, firstPeriod,
                                        salvage, period, rate, dcb);
        Console.Out.WriteLine("The depreciation for the second accounting "+ "period is "+ amorlinc);
    }
}
```

Output

The depreciation for the second accounting period is 360

Example: Convexity for a Security

The convexity of a 10 year bond which pays interest semiannually is returned in this example.

```csharp
using System;
using Imsl.Finance;

public class convexityEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("7/1/90");
        DateTime maturity = DateTime.Parse("7/1/00");
        double coupon = .075;
        double yield = .09;
        Bond.Frequency freq = Bond.Frequency.SemiAnnual;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        double convexity = Bond.Convexity(settlement, maturity, coupon,
                                           yield, freq, dcb);
        Console.Out.WriteLine("The convexity of the bond with semiannual "+ "interest payments is "+ convexity);
    }
}
```
The convexity of the bond with semiannual interest payments is 59.4049912915856

Example: Days - Beginning of Period to Settlement Date

In this example, the settlement date is 11/11/86. The number of days from the beginning of the coupon period to the settlement date is returned.

```csharp
using System;
using Imsl.Finance;

class coupdaybsEx1
{
    public static void Main(string[] args)
    {
        DateTime settlement = DateTime.Parse("11/11/86");
        DateTime maturity = DateTime.Parse("3/1/99");
        Bond.Frequency freq = Bond.Frequency.SemiAnnual;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        int coupdaybs = Bond.Coupdaybs(settlement, maturity, freq, dcb);
        Console.Out.WriteLine("The number of days from the beginning of the" + "
coupon period to the settlement date is " + coupdaybs);
    }
}
```

Output

The number of days from the beginning of the coupon period to the settlement date is 71

Example: Days in the Settlement Date Period

In this example, the settlement date is 11/11/86. The number of days in the coupon period containing this date is returned.

```csharp
using System;
using Imsl.Finance;

class coupdaysEx1
{
    public static void Main(string[] args)
    {
        // Code here
    }
}
```
Output

The number of days in the coupon period that contains the settlement date is 182.5

Example: Days - Settlement Date to Next Coupon Date

In this example, the settlement date is 11/11/86. The number of days from this date to the next coupon date is returned.

using System;
using Imsl.Finance;

public class coupdaysncEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("11/11/86");
        DateTime maturity = DateTime.Parse("3/1/99");
        Bond.Frequency freq = Bond.Frequency.SemiAnnual;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        int coupdaysnc = Bond.Coupdaysnc(settlement, maturity, freq, dcb);
        Console.Out.WriteLine("The number of days from the settlement date "
                          + "to the next coupon date is " + coupdaysnc);
    }
}

Output

The number of days from the settlement date to the next coupon date is 110
Example: Next Coupon Date After the Settlement Date

In this example, the settlement date is 11/11/86. The previous coupon date before this date is returned.

```csharp
using System;
using Imsl.Finance;

public class coupncdEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("11/11/86");
        DateTime maturity = DateTime.Parse("3/1/99");
        Bond.Frequency freq = Bond.Frequency.SemiAnnual;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        DateTime coupncd = Bond.Coupncd(settlement, maturity, freq, dcb);
        Console.Out.WriteLine("The next coupon date after the " +
                             "settlement date is " + coupncd);
    }
}
```

Output

The next coupon date after the settlement date is 3/1/1987 12:00:00 AM

Example: Number of Payable Coupons

In this example, the settlement date is 11/11/86. The number of payable coupons between this date and the maturity date is returned.

```csharp
using System;
using Imsl.Finance;

public class coupnumEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("11/11/86");
        DateTime maturity = DateTime.Parse("3/1/99");
        Bond.Frequency freq = Bond.Frequency.SemiAnnual;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        int coupnum = Bond.Coupnum(settlement, maturity, freq, dcb);
        Console.Out.WriteLine("The number of coupons payable between" +
                             " the \nsettlement date and the maturity" +
                             " date is " + coupnum);
    }
}
```
Output

The number of coupons payable between the settlement date and the maturity date is 25

Example: Previous Coupon Date Before the Settlement Date

In this example, the settlement date is 11/11/86. The previous coupon date before this date is returned.

```csharp
using System;
using Imsl.Finance;

class couppcdEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("11/11/86");
        DateTime maturity = DateTime.Parse("3/1/99");
        Bond.Frequency freq = Bond.Frequency.SemiAnnual;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        DateTime couppcd = Bond.Couppcd(settlement, maturity, freq, dcb);
        Console.Out.WriteLine("The previous coupon date before the 
settlement date is " +
                      couppcd.ToLongDateString());
    }
}
```

Output

The previous coupon date before the settlement date is Monday, September 01, 1986

Example: Discount Rate for a Security

In this example, the discount rate for a security is returned.

```csharp
using System;
using Imsl.Finance;

class discEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("2/15/92");
        DateTime maturity = DateTime.Parse("6/10/92");
        double price = 97.975;
```
double redemption = 100.0;
DayCountBasis dcb = DayCountBasis.BasisActual365;
double disc = Bond.Disc(settlement, maturity, price,
redeemption, dcb);
Console.Out.WriteLine("The discount rate for the security is "+ disc);
}

Output

The discount rate for the security is 0.0637176724137933

Example: Duration of a Security with Periodic Payments

The annual duration of a 10 year bond which pays interest semiannually is returned in this example.

using System;
using Imsl.Finance;

class durationEx1
{
    public static void Main(string[] args)
    {
        DateTime settlement = DateTime.Parse("7/1/85");
        DateTime maturity = DateTime.Parse("7/1/95");
        double coupon = .075;
        double yield = .09;
        Bond.Frequency freq = Bond.Frequency.SemiAnnual;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        double duration = Bond.Duration(settlement, maturity, coupon,
        yield, freq, dcb);
        Console.Out.WriteLine("The annual duration of the bond with" +
        "\nsemiannual interest payments is "+
duration);
    }
}

Output

The annual duration of the bond with
semiannual interest payments is 7.04195337797215

Example: Interest Rate of a Fully Invested Security

The discount rate of a 10 year bond is returned in this example.
using System;
using Imsl.Finance;

public class intrateEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("7/1/85");
        DateTime maturity = DateTime.Parse("7/1/95");
        double investment = 7000.0;
        double redemption = 10000.0;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        double intrate = Bond.Intrate(settlement, maturity, investment,
                                      redemption, dcb);
        Console.Out.WriteLine("The interest rate of the bond is " +
                              intrate);
    }
}

Output

The interest rate of the bond is 0.0428336723517446

Example: Modified Macauley Duration of a Security with Periodic Payments

The modified Macauley duration of a 10 year bond which pays interest semiannually is returned in this example.

using System;
using Imsl.Finance;

public class mdurationEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("7/1/85");
        DateTime maturity = DateTime.Parse("7/1/95");
        double coupon = .075;
        double yield = .09;
        Bond.Frequency freq = Bond.Frequency.SemiAnnual;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        double mduration = Bond.Mduration(settlement, maturity, coupon,
                                            yield, freq, dcb);
        Console.Out.WriteLine("The modified Macauley duration " +
                              "of the bond is " + mduration);
    }
}
Output

The modified Macauley duration of the bond with semiannual interest payments is 6.73871136648053

Example: Price of a Security

The price per $100 face value of a 10 year bond which pays interest semiannually is returned in this example.

```csharp
using System;
using Imsl.Finance;

class priceEx1
{
    public static void Main(string[] args)
    {
        DateTime settlement = DateTime.Parse("7/1/85");
        DateTime maturity = DateTime.Parse("7/1/95");
        double rate = .06;
        double yield = .07;
        double redemption = 105.0;
        Bond.Frequency freq = Bond.Frequency.SemiAnnual;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double price = Bond.Price(settlement, maturity, rate, yield,
                                  redemption, freq, dcb);
        Console.Out.WriteLine("The price of the bond is " +
                            price.ToString("C");
    }
}
```

Output

The price of the bond is $95.41

Example: Price of a Discounted Security

The price per $100 face value of a discounted 1 year bond is returned in this example.

```csharp
using System;
using Imsl.Finance;

class pricediscEx1
{
    public static void Main(string[] args)
    {
        DateTime settlement = DateTime.Parse("7/1/85");
        DateTime maturity = DateTime.Parse("7/1/86");
```
double rate = .05;
double redemption = 100.0;
DayCountBasis dcb = DayCountBasis.BasisNASD;
double pricedisc = Bond.Pricedisc(settlement, maturity, rate, 
redemption, dcb);
Console.Out.WriteLine("The price of the discounted bond is " +
pricedisc.ToString("C"));
}
}

Output
The price of the discounted bond is $95.00

Example: Price of a Security that Pays at Maturity

The price per $100 face value of 1 year bond that pays interest at maturity is returned in this example.

using System;
using Imsl.Finance;

public class pricematEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("8/1/85");
        DateTime maturity = DateTime.Parse("7/1/86");
        DateTime issue = DateTime.Parse("7/1/85");
        double rate = .05;
        double yield = .05;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double pricemat = Bond.Pricemat(settlement, maturity, issue,
                                         rate, yield, dcb);
        Console.Out.WriteLine("The price of the bond is " + pricemat);
    }
}

Output
The price of the bond is 99.9817397078353

Price of a Discounted Security

The price of a discounted 1 year bond is returned in this example.
priceyieldEx1

using System;
using Imsl.Finance;

public class priceyieldEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("7/1/85");
        DateTime maturity = DateTime.Parse("7/1/95");
        double yield = 0.010055244588347783;
        double redemption = 105.0;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double priceyield = Bond.Priceyield(settlement, maturity,
                                             yield, redemption, dcb);
        Console.Out.WriteLine("The price of the discounted bond is "+
                              priceyield);
    }
}

Output

The price of the discounted bond is 95.40663

Example: Amount Received at Maturity for a Fully Invested Security

The amount to be received at maturity for a 10 year bond is returned in this example.

using System;
using Imsl.Finance;

public class receivedEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("7/1/85");
        DateTime maturity = DateTime.Parse("7/1/95");
        double investment = 7000.0;
        double discount = .06;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        double received = Bond.Received(settlement, maturity,
                                        investment, discount, dcb);
        Console.Out.WriteLine("The amount received at maturity for the"+
                              " bond is " + received.ToString("C"));
    }
}
Output

The amount received at maturity for the bond is $17,514.40

Example: Bond-Equivalent Yield

The bond-equivalent yield for a 1 year Treasury bill is returned in this example.

```csharp
using System;
using Imsl.Finance;

class tbilleqEx1
{
    static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("7/1/85");
        DateTime maturity = DateTime.Parse("7/1/86");
        double discount = .05;
        double tbilleq = Bond.Tbilleq(settlement, maturity, discount);
        Console.Out.WriteLine("The bond-equivalent yield for the " +
                             "T-bill is " + tbilleq.ToString("P"));
    }
}
```

Output

The bond-equivalent yield for the T-bill is 5.27 %

Example: Treasury Bill Price

The price per $100 face value for a 1 year Treasury bill is returned in this example.

```csharp
using System;
using Imsl.Finance;

class tbillpriceEx1
{
    static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("7/1/85");
        DateTime maturity = DateTime.Parse("7/1/86");
        double discount = .05;
        double tbillprice = Bond.Tbillprice(settlement, maturity, discount);
        Console.Out.WriteLine("The price per $100 face value for the " +
                             "T-bill is " + tbillprice.ToString("C");
    }
}
```
Output

The price per $100 face value for the T-bill is $94.93

Example: Treasury Bill Yield

The yield for a 1 year Treasury bill is returned in this example.

```csharp
using System;
using Imsl.Finance;

class tbillyieldEx1
{
    static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("7/1/85");
        DateTime maturity = DateTime.Parse("7/1/86");
        double price = 94.93;
        double tbillyield = Bond.Tbillyield(settlement, maturity, price);
        Console.WriteLine("The yield for the T-bill is " + tbillyield.ToString("P"));
    }
}
```

Output

The yield for the T-bill is 5.27%

Example: Year Fraction

The year fraction of a 30/360 year starting 8/1/85 and ending 7/1/86 is returned in this example.

```csharp
using System;
using Imsl.Finance;

class yearfracEx1
{
    static void Main(String[] args)
    {
        DateTime start = DateTime.Parse("8/1/85");
        DateTime end = DateTime.Parse("7/1/86");
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double yearfrac = Bond.Yearfrac(start, end, dcb);
        Console.WriteLine("The year fraction of the 30/360 period is " + yearfrac);
    }
}
```
**Output**

The year fraction of the 30/360 period is 0.916666666666667

**Example: Yield on a Security**

The yield on a 10 year bond which pays interest semiannually is returned in this example.

```csharp
using System;
using Imsl.Finance;

public class yieldEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("7/1/85");
        DateTime maturity = DateTime.Parse("7/1/95");
        double rate = .06;
        double price = 95.40663;
        double redemption = 105.0;
        Bond.Frequency freq = Bond.Frequency.SemiAnnual;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double yield = Bond.Yield(settlement, maturity, rate, price,
                                  redemption, freq, dcb);
        Console.Out.WriteLine("The yield of the bond is "+ yield);
    }
}
```

**Output**

The yield of the bond is 0.0699999968284289

**Example: Yield on a Discounted Security**

The yield on a discounted 10 year bond is returned in this example.

```csharp
using System;
using Imsl.Finance;

public class yielddiscEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("7/1/85");
        DateTime maturity = DateTime.Parse("7/1/95");
        double price = 95.40663;
        double redemption = 105.0;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
```

---

*Finance*  
*Bond Class • 655*
double yielddisc = Bond.Yielddisc(settlement, maturity, price, 
redemption, dcb);
Console.Out.WriteLine("The yield on the discounted bond is " + 
yielddisc);
}

Output

The yield on the discounted bond is 0.0100552445883478

**Example: Yield on a Security Which Pays at Maturity**

The yield on a bond which pays at maturity is returned in this example.

using System;
using Imsl.Finance;

public class yieldmatEx1
{
    public static void Main(String[] args)
    {
        DateTime settlement = DateTime.Parse("8/1/85");
        DateTime maturity = DateTime.Parse("7/1/95");
        DateTime issue = DateTime.Parse("7/1/85");
        double rate = .06;
        double price = 95.40663;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double yieldmat = Bond.Yieldmat(settlement, maturity, issue, 
            rate, price, dcb);
        Console.Out.WriteLine("The yield on a bond which pays at " + 
            "maturity is " + yieldmat);
    }
}

Output

The yield on a bond which pays at maturity is 0.0673905127809195

**Bond.Frequency Enumeration**

Frequency of the bond’s coupon payments.
public enumeration Imsl.Finance.Bond.Frequency

Fields

Annual
  public Imsl.Finance.Bond.Frequency Annual
  Indicates interest is paid once a year.

Quarterly
  public Imsl.Finance.Bond.Frequency Quarterly
  Indicates interest is paid four times a year.

SemiAnnual
  public Imsl.Finance.Bond.Frequency SemiAnnual
  Indicates interest is paid twice a year.

DayCountBasis Class

The Day Count Basis.

public class Imsl.Finance.DayCountBasis

Fields

Basis30e360
  public Imsl.Finance.DayCountBasis Basis30e360
  Computations based on the assumption of 30 days per month and 360 days per year.
  See Also: Imsl.Finance.DayCountBasis.BasisPart30E360 (p. 658)

BasisActual360
  public Imsl.Finance.DayCountBasis BasisActual360
  Computations are based on the number of days in a month based on the actual calendar value and the number of days, but assuming 360 days per year.

BasisActual365
  public Imsl.Finance.DayCountBasis BasisActual365
  Computations are based on the number of days in a month based on the actual calendar value and the number of days, but assuming 365 days per year.
BasisActual
public Imsl.Finance.DayCountBasis BasisActual
Computations are based on the actual calendar.
See Also: Imsl.Finance.DayCountBasis.BasisPartActual (p. 658)

BasisNASD
public Imsl.Finance.DayCountBasis BasisNASD
Computations based on the assumption of 30 days per month and 360 days per year.
See Also: Imsl.Finance.DayCountBasis.BasisPartNASD (p. 658)

BasisPart30E360
public Imsl.Finance.IBasisPart BasisPart30E360
Computations based on the assumption of 30 days per month and 360 days per year. This computes the number of days between two dates differently than BasisPartNASD for months with other than 30 days.

BasisPart365
public Imsl.Finance.IBasisPart BasisPart365
Computations based on the assumption of 365 days per year.

BasisPartActual
public Imsl.Finance.IBasisPart BasisPartActual
Computations are based on the actual calendar.

BasisPartNASD
public Imsl.Finance.IBasisPart BasisPartNASD
Computations based on the assumption of 30 days per month and 360 days per year.

Properties

MonthBasis
public Imsl.Finance.IBasisPart MonthBasis {get; }
The (days in month) portion of the Day Count Basis. A IBasisPart object which represents the month Basis for this DayCountBasis.

YearBasis
public Imsl.Finance.IBasisPart YearBasis {get; }
The (days in year) portion of the Day Count Basis. A IBasisPart object which represents the year Basis for this DayCountBasis.

Constructor

DayCountBasis
public DayCountBasis(Imsl.Finance.IBasisPart monthBasis,
Imsl.Finance.IBasisPart yearBasis)
Creates a new DayCountBasis.
**monthBasis**  A IBasisPart which specifies the month basis.

**yearBasis**  A IBasisPart which specifies the year basis.

**Description**

Rules for computing the number or days between two dates or number of days in a year. For many securities, computations are based on rules other than on the actual calendar.
Chapter 23: Neural Nets

Types

class FeedForwardNetwork ................................................................. 688
class Layer .................................................................................. 699
class InputLayer ............................................................................. 700
class HiddenLayer .......................................................................... 701
class OutputLayer ........................................................................... 702
class Node ......................................................................................... 704
class InputNode ................................................................................ 705
class Perceptron ............................................................................... 706
class OutputPerceptron ................................................................. 707
class IActivation ............................................................................... 708
class Link .......................................................................................... 709
class ITrainer .................................................................................... 710
class QuasiNewtonTrainer ................................................................. 711
interface QuasiNewtonTrainer.IError ................................................ 715
class LeastSquaresTrainer ................................................................. 716
class EpochTrainer ............................................................................ 719
class ScaleFilter ............................................................................... 721
enumeration ScaleFilter.ScalingMethod .......................................... 729
class UnsupervisedNominalFilter ..................................................... 730
class UnsupervisedOrdinalFilter ......................................................... 733
enumeration UnsupervisedOrdinalFilter.TransformMethod ................ 738
class TimeSeriesFilter ...................................................................... 738
class TimeSeriesClassFilter .............................................................. 741
class Network ..................................................................................... 762

Usage Notes

Neural Networks - An Overview

Today, neural networks are used to solve a wide variety of problems, some of which have been
solved by existing statistical methods, and some of which have not. These applications fall into one of the following three categories:

- **Forecasting**: predicting one or more quantitative outcomes from both quantitative and categorical input data,
- **Classification**: classifying input data into one of two or more categories, or
- **Statistical pattern recognition**: uncovering patterns, typically spatial or temporal, among a set of variables.

Forecasting, pattern recognition and classification problems are not new. They existed years before the discovery of neural network solutions in the 1980’s. What is new is that neural networks provide a single framework for solving so many traditional problems and, in some cases, extend the range of problems that can be solved.

Traditionally, these problems have been solved using a variety of well known statistical methods:

- linear regression and general least squares,
- logistic regression and discrimination,
- principal component analysis,
- discriminant analysis,
- \(k\)-nearest neighbor classification, and
- ARMA and non-linear ARMA time series forecasts.

In many cases, simple neural network configurations yield the same solution as many traditional statistical applications. For example, a single-layer, feed-forward neural network with linear activation for its output perceptron is equivalent to a general linear regression fit. Neural networks can provide more accurate and robust solutions for problems where traditional methods do not completely apply.

Mandic and Chambers (2001) point out that traditional methods for time series forecasting are unsuitable when a time series:

- is non-stationary,
- has large amounts of noise, such as a biomedical series, or
- is too short.

ARIMA and other traditional time series approaches can produce poor forecasts when one or more of the above conditions exist. The forecasts of ARMA and non-linear ARMA (NARMA) depend heavily upon key assumptions about the model or underlying relationship between the output of the series and its patterns.
Neural networks, on the other hand, adapt to changes in a non-stationary series and can produce reliable forecasts even when the series contains a good deal of noise or when only a short series is available for training. Neural networks provide a single tool for solving many problems traditionally solved using a wide variety of statistical tools and for solving problems when traditional methods fail to provide an acceptable solution.

Although neural network solutions to forecasting, pattern recognition, and classification problems can be very different, they are always the result of computations that proceed from the network inputs to the network outputs. The network inputs are referred to as patterns, and outputs are referred to as classes. Frequently the flow of these computations is in one direction, from the network input patterns to its outputs. Networks with forward-only flow are referred to as feed-forward networks.

![Figure 1. A 2-layer, Feed-Forward Network with 4 Inputs and 2 Outputs](image)

Other networks, such as recurrent neural networks, allow data and information to flow in both directions, see Mandic and Chambers (2001).
A neural network is defined not only by its architecture and flow, or interconnections, but also by computations used to transmit information from one node or input to another node. These computations are determined by network weights. The process of fitting a network to existing data to determine these weights is referred to as training the network, and the data used in this process are referred to as patterns. Individual network inputs are referred to as attributes and outputs are referred to as classes. Many terms used to describe neural networks are synonymous to common statistical terminology.

Table 1. Synonyms between Neural Network and Common Statistical Terminology
Neural Networks – History and Terminology

The Threshold Neuron

McCulloch and Pitts (1943) wrote one of the first published works on neural networks. In their paper, they describe the threshold neuron as a model for how the human brain stores and processes information.

The McCulloch and Pitts Threshold Neuron

All inputs to a threshold neuron are combined into a single number, $Z$, using the following weighted sum:

$$Z = \sum_{i=1}^{m} w_i x_i - \mu$$
where $w_i$ is the weight associated with the $i$-th input (attribute) $x_i$. The term $\mu$ in this calculation is referred to as the bias term. In traditional statistical terminology, it might be referred to as the intercept. The weights and bias terms in this calculation are estimated during network training.

In McCulloch and Pitt’s description of the threshold neuron, the neuron does not respond to its inputs unless $Z$ is greater than zero. If $Z$ is greater than zero then the output from this neuron is set to 1. If $Z$ is less than zero the output is zero:

$$Y = \begin{cases} 1 & \text{if } Z > 0 \\ 0 & \text{if } Z \leq 0 \end{cases}$$

where $Y$ is the neuron’s output.

For years following their 1943 paper, interest in the McCulloch and Pitts neural network was limited to theoretical discussions, such as those of Hebb (1949), about learning, memory, and the brain’s structure.

**The Perceptron**

The McCulloch and Pitts neuron is also referred to as a threshold neuron since it abruptly changes its output from 0 to 1 when its potential, $Z$, crosses a threshold. Mathematically, this behavior can be viewed as a step function that maps the neuron’s potential, $Z$, to the neuron’s output, $Y$.

Rosenblatt (1958) extended the McCulloch and Pitts threshold neuron by replacing this step function with a continuous function that maps $Z$ to $Y$. The Rosenblatt neuron is referred to as the perceptron, and the continuous function mapping $Z$ to $Y$ makes it easier to train a network of perceptrons than a network of threshold neurons.

Unlike the threshold neuron, the perceptron produces analog output rather than the threshold neuron’s purely binary output. Carefully selecting the analog function makes Rosenblatt’s perceptron differentiable, whereas the threshold neuron is not. This simplifies the training algorithm.

Like the threshold neuron, Rosenblatt’s perceptron starts by calculating a weighted sum of its inputs, $Z = \sum_{i=1}^{n} w_i x_i - \mu$. This is referred to as the perceptron’s potential.

Rosenblatt’s perceptron calculates its analog output from its potential. There are many choices for this calculation. The function used for this calculation is referred to as the activation function in Figure 4 below.
As shown in Figure 4, perceptrons consist of the following five components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>$X_1, X_2, X_3$</td>
</tr>
<tr>
<td>Input Weights</td>
<td>$W_1, W_2, W_3$</td>
</tr>
<tr>
<td>Potential</td>
<td>$Z = \sum_{i=1}^{3} W_i X_i - \mu$, where $\mu$ is a bias correction.</td>
</tr>
<tr>
<td>Activation Function</td>
<td>$g(Z)$</td>
</tr>
<tr>
<td>Output</td>
<td>$g(Z)$</td>
</tr>
</tbody>
</table>

Like threshold neurons, perceptron inputs can be either the initial raw data inputs or the output from another perceptron. The primary purpose of the network training is to estimate the weights associated with each perceptron’s potential. The activation function maps this potential to the perceptron’s output.

**The Activation Function**

Although theoretically any differential function can be used as an activation function, the identity and sigmoid functions are the two most commonly used.

The *identity activation* function, also referred to as a *linear activation* function, is a flow-through mapping of the perceptron’s potential to its output:

$$g(Z) = Z$$

Output perceptrons in a forecasting network often use the identity activation function.
Figure 5. An Identity (Linear) Activation Function

If the identity activation function is used throughout the network, then it is easily shown that the network is equivalent to fitting a linear regression model of the form

\[ Y_i = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k, \]

where \( x_1, x_2, \ldots, x_k \) are the \( k \) network inputs, \( Y_i \) is the \( i \)-th network output and \( \beta_0, \beta_1, \ldots, \beta_k \) are the coefficients in the regression equation. As a result, it is uncommon to find a neural network with identity activation used in all its perceptrons.

_Sigmoid activation_ functions are differentiable functions that map the perceptron’s potential to a range of values, such as 0 to 1, i.e., \( \mathbb{R}^K \rightarrow \mathbb{R} \) where \( K \) is the number of perceptron inputs.
In practice, the most common sigmoid activation function is the logistic function that maps the potential into the range 0 to 1:

\[ g(Z) = \frac{1}{1 + e^{-Z}} \]

Since \( 0 < g(Z) < 1 \), the logistic function is very popular for use in networks that output probabilities.

Other popular sigmoid activation functions include:

- the hyperbolic-tangent \( g(Z) = \tanh(Z) = \frac{e^{\alpha Z} - e^{-\alpha Z}}{e^{\alpha Z} + e^{-\alpha Z}} \)
- the arc-tangent \( g(Z) = \frac{2}{\pi} \arctan \left( \frac{\pi Z}{2} \right) \), and
- the squash activation function (Elliott (1993)) \( g(Z) = \frac{Z}{1 + |Z|} \)

It is easy to show that the hyperbolic-tangent and logistic activation functions are linearly related. Consequently, forecasts produced using logistic activation should be close to those produced using hyperbolic-tangent activation. However, one function may be preferred over the other when training performance is a concern. Researchers report that the training time using the hyperbolic-tangent activation function is shorter than using the logistic activation function.
Network Applications

Forecasting using Neural Networks

There are many good statistical forecasting tools. Most require assumptions about the relationship between the variables being forecasted and the variables used to produce the forecast, as well as the distribution of forecast errors. Such statistical tools are referred to as parametric methods. ARIMA time series models, for example, assume that the time series is stationary, that the errors in the forecasts follow a particular ARIMA model, and that the probability distribution for the residual errors is Gaussian, see Box and Jenkins (1970). If these assumptions are invalid, then ARIMA time series forecasts can be very poor.

Neural networks, on the other hand, require few assumptions. Since neural networks can approximate highly non-linear functions, they can be applied without an extensive analysis of underlying assumptions.

Another advantage of neural networks over ARIMA modeling is the number of observations needed to produce a reliable forecast. ARIMA models generally require 50 or more equally spaced, sequential observations in time. In many cases, neural networks can also provide adequate forecasts with fewer observations by incorporating exogenous, or external, variables in the network’s input.

For example, a company applying ARIMA time series analysis to forecast business expenses would normally require each of its departments, and each sub-group within each department to prepare its own forecast. For large corporations this can require fitting hundreds or even thousands of ARIMA models. With a neural network approach, the department and sub-group information could be incorporated into the network as exogenous variables. Although this can significantly increase the network’s training time, the result would be a single model for predicting expenses within all departments and sub-departments.

Linear least squares models are also popular statistical forecasting tools. These methods range from simple linear regression for predicting a single quantitative outcome to logistic regression for estimating probabilities associated with categorical outcomes. It is easy to show that simple linear least squares forecasts and logistic regression forecasts are equivalent to a feed-forward network with a single layer. For this reason, single-layer feed-forward networks are rarely used for forecasting. Instead multilayer networks are used.

Hutchinson (1994) and Masters (1995) describe using multilayer feed-forward neural networks for forecasting. Multilayer feed-forward networks are characterized by the forward-only flow of information in the network. The flow of information and computations in a feed-forward network is always in one direction, mapping an M-dimensional vector of inputs to a C-dimensional vector of outputs, i.e., \( \mathbb{R}^M \rightarrow \mathbb{R}^C \).

There are many other types of networks without this feed-forward requirement. Information and computations in a recurrent neural network, for example, flows in both directions. Output from one level of a recurrent neural network can be fed back, with some delay, as input into the same network, see Figure 2. Recurrent networks are very useful for time series prediction, see Mandic and Chambers (2001).
Pattern Recognition using Neural Networks

Neural networks are also extensively used in statistical pattern recognition. Pattern recognition applications that make wide use of neural networks include:

- natural language processing: Manning and Schütze (1999)
- speech and text recognition: Lippmann (1989)
- face recognition: Lawrence, et al. (1997)
- playing backgammon, Tesauro (1990)

The interest in pattern recognition using neural networks has stimulated the development of important variations of feed-forward networks. Two of the most popular are:

- Self-Organizing Maps, also called Kohonen Networks, Kohonen (1995),

Good mathematical descriptions of the neural network methods underlying these applications are given by Bishop (1995), Ripley (1996), Mandic and Chambers (2001), and Abe (2001). An excellent overview of neural networks, from a statistical viewpoint, is also found in Warner and Misra (1996).

Neural Networks for Classification

Classifying observations using prior concomitant information is possibly the most popular application of neural networks. Data classification problems abound in business and research. When decisions based upon data are needed, they can often be treated as a neural network data classification problem. Decisions to buy, sell, hold or do nothing with a stock, are decisions involving four choices. Classifying loan applicants as good or bad credit risks, based upon their application, is a classification problem involving two choices. Neural networks are powerful tools for making decisions or choices based upon data.

These same tools are ideally suited for automatic selection or decision-making. Incoming email, for example, can be examined to separate spam from important email using a neural network trained for this task. A good overview of solving classification problems using multilayer feed-forward neural networks is found in Abe (2001) and Bishop (1995).

There are two popular methods for solving data classification problems using multilayer feed-forward neural networks, depending upon the number of choices (classes) in the classification problem. If the classification problem involves only two choices, then it can be solved using a neural network with one logistic output. This output estimates the probability that the input data belong to one of the two choices.
For example, a multilayer feed-forward network with a single, logistic output can be used to determine whether a new customer is credit-worthy. The network’s input would consist of information on the applicants credit application, such as age, income, etc. If the network output probability is above some threshold value (such as 0.5 or higher) then the applicant’s credit application is approved.

This is referred to as binary classification using a multilayer feed-forward neural network. If more than two classes are involved then a different approach is needed. A popular approach is to assign logistic output perceptrons to each class in the classification problem. The network assigns each input pattern to the class associated with the output perceptron that has the highest probability for that input pattern. However, this approach produces invalid probabilities since the sum of the individual class probabilities for each input is not equal to one, which is a requirement for any valid multivariate probability distribution.

To avoid this problem, the softmax activation function, see Bridle (1990), applied to the network outputs ensures that the outputs conform to the mathematical requirements of multivariate classification probabilities. If the classification problem has C categories, or classes, then each category is modeled by one of the network outputs. If \( Z_i \) is the weighted sum of products between its weights and inputs for the \( i \)-th output, i.e., \( Z_i = \sum_j w_{ji} y_{ji} \), then

\[
\text{softmax}_i = \frac{e^{Z_i}}{\sum_{j=1}^{C} e^{Z_j}}
\]

The softmax activation function ensures that the outputs all conform to the requirements for multivariate probabilities. That is,

\[ 0 < \text{softmax}_i < 1, \quad \text{for all } i = 1, 2, \ldots, C \]

and

\[ \sum_{i=1}^{C} \text{softmax}_i = 1 \]

A pattern is assigned to the \( i \)-th classification when \( \text{softmax}_i \) is the largest among all \( C \) classes. However, multilayer feed-forward neural networks are only one of several popular methods for solving classification problems. Others include:

- Support Vector Machines (SVM Neural Networks), Abe (2001),
- Classification and Regression Trees (CART), Breiman, et al. (1984),
- Quinlan’s classification algorithms C4.5 and C5.0, Quinlan (1993), and
- Quick, Unbiased and Efficient Statistical Trees (QUEST), Loh and Shih (1997).

Support Vector Machines are simple modifications of traditional multilayer feed-forward neural networks (MLFF) configured for pattern classification.
Multilayer Feed-Forward Neural Networks

A multilayer feed-forward neural network is an interconnection of perceptrons in which data and calculations flow in a single direction, from the input data to the outputs. The number of layers in a neural network is the number of layers of perceptrons. The simplest neural network is one with a single input layer and an output layer of perceptrons. The network in Figure 7 illustrates this type of network. Technically this is referred to as a one-layer feed-forward network with two outputs because the output layer is the only layer with an activation calculation.

![Diagram of a single-layer feed-forward neural network](image)

**Figure 7. A Single-Layer Feed-Forward Neural Net**

In this single-layer feed-forward neural network, the networks inputs are directly connected to the output layer perceptrons, $Z_1$ and $Z_2$.

The output perceptrons use activation functions, $g_1$ and $g_2$, to produce the outputs $Y_1$ and $Y_2$. Since

$$Z_1 = \sum_{i=1}^{3} W_{1,i}X_i - \mu_1$$

and

$$Z_2 = \sum_{i=1}^{3} W_{2,i}X_i - \mu_2$$

$$Y_1 = g_1(Z_1) = g_1(\sum_{i=1}^{3} W_{1,i}X_i - \mu_1)$$
and

\[ Y_2 = g_2(Z_2) = g_2(\sum_{i=1}^{3} W_{2,i}X_i - \mu_2) \]

When the activation functions \( g_1 \) and \( g_2 \) are identity activation functions, a single-layer neural net is equivalent to a linear regression model. Similarly, if \( g_1 \) and \( g_2 \) are logistic activation functions, then the single-layer neural net is equivalent to logistic regression. Because of this correspondence between single-layer neural networks and linear and logistic regression, single-layer neural networks are rarely used in place of linear and logistic regression.

The next most complicated neural network is one with two layers. This extra layer is referred to as a hidden layer. In general there is no restriction on the number of hidden layers. However, it has been shown mathematically that a two-layer neural network, such as shown in Figure 1, can accurately reproduce any differentiable function, provided the number of perceptrons in the hidden layer is unlimited.

However, increasing the number of neurons increases the number of weights that must be estimated in the network, which in turn increases the execution time for this network. Instead of increasing the number of perceptrons in the hidden layers to improve accuracy, it is sometimes better to add additional hidden layers, which typically reduces both the total number of network weights and the computational time. However, in practice, it is uncommon to see neural networks with more than two or three hidden layers.

**Neural Network Error Calculations**

**Error Calculations for Forecasting**

The error calculations used to train a neural network are very important. Researchers have investigated many error calculations, trying to find a calculation with a short training time that is appropriate for the network’s application. Typically error calculations are very different depending primarily on the network’s application.

For forecasting, the most popular error function is the sum-of-squared errors, or one of its scaled versions. This is analogous to using the minimum least squares optimization criterion in linear regression. Like least squares, the sum-of-squared errors is calculated by looking at the squared difference between what the network predicts for each training pattern and the target value, or observed value, for that pattern. Formally, the equation is the same as one-half the traditional least squares error:

\[
E = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{C} (t_{ij} - \hat{t}_{ij})^2
\]

where \( N \) is the total number of training cases, \( C \) is equal to the number of network outputs, \( t_{ij} \) is the observed output for the \( i \)-th training case and the \( j \)-th network output, and \( \hat{t}_{ij} \) is the network’s forecast for that case.

Common practice recommends fitting a different network for each forecast variable. That is, the recommended practice is to use \( C=1 \) when using a multilayer feed-forward neural network.
for forecasting. For classification problems with more than two classes, it is common to associate one output with each classification category, i.e., \( C = \text{number of classes} \).

Notice that in ordinary least squares, the sum-of-squared errors are not multiplied by one-half. Although this has no impact on the final solution, it significantly reduces the number of computations required during training.

Also note that as the number of training patterns increases, the sum-of-squared errors increases. As a result, it is often useful to use the root-mean-square (RMS) error instead of the unscaled sum-of-squared errors:

\[
E_{\text{RMS}} = \frac{\sum_{i=1}^{N} \sum_{j=1}^{C} (t_{ij} - \hat{t}_{ij})^2}{\sum_{i=1}^{N} \sum_{j=1}^{C} (t_{ij} - \bar{t})^2}
\]

where \( \bar{t} \) is the average output:

\[
\bar{t} = \frac{\sum_{i=1}^{N} \sum_{j=1}^{C} t_{ij}}{N \cdot C}
\]

Unlike the unscaled sum-of-squared errors, \( E_{\text{RMS}} \) does not increase as \( N \) increases. The smaller the value of \( E_{\text{RMS}} \) the closer the network is predicting its targets during training. A value of \( E_{\text{RMS}} = 0 \) indicates that the network is able to predict every pattern exactly. A value of \( E_{\text{RMS}} = 1 \) indicates that the network is predicting the training cases only as well as using the mean of the training cases for forecasting.

Notice that the root-mean-squared error is related to the sum-of-squared error by a simple scale factor:

\[
E_{\text{RMS}} = \frac{2}{\bar{t}} \cdot E
\]

Another popular error calculation for forecasting from a neural network is the Minkowski-R error. The sum-of-squared error, \( E \), and the root-mean-squared error, \( E_{\text{RMS}} \), are both theoretically motivated by assuming the noise in the target data is Gaussian. In many cases, this assumption is invalid. A generalization of the Gaussian distribution to other distributions gives the following error function, referred to as the Minkowski-R error:

\[
E_{R} = \sum_{i=1}^{N} \sum_{j=1}^{C} |t_{ij} - \hat{t}_{ij}|^R.
\]

Notice that \( E_{R} = 2E \) when \( R = 2 \).

A good motivation for using \( E_{R} \) instead of \( E \) is to reduce the impact of outliers in the training data. The usual error measures, \( E \) and \( E_{\text{RMS}} \), emphasize larger differences between the training data and network forecasts since they square those differences. If outliers are expected, then it is better to de-emphasize larger differences. This can be done by using the Minkowski-R error.
with R=1. When R=1, the Minkowski-R error simplifies to the sum of absolute differences:

\[ L = E^1 = \sum_{i=1}^{N} \sum_{j=1}^{C} |t_{ij} - \hat{t}_{ij}|. \]

\( L \) is also referred to as the Laplacian error. Its name is derived from the fact that it can be theoretically justified by assuming the noise in the training data follows a Laplacian rather than Gaussian distribution.

Of course, similar to \( E \), \( L \) generally increases when the number of training cases increases. Similar to \( E^{RMS} \), a scaled version of the Laplacian error can be calculated using the following formula:

\[ L^{RMS} = \frac{\sum_{i=1}^{N} \sum_{j=1}^{C} |t_{ij} - \hat{t}_{ij}|}{\sum_{i=1}^{N} \sum_{j=1}^{C} |t_{ij} - \bar{t}|}. \]

**Cross-Entropy Error for Binary Classification**

As previously mentioned, multilayer feed-forward neural networks can be used for both forecasting and classification applications. Training a forecasting network involves finding the network weights that minimize either the Gaussian or Laplacian distributions, \( E \) or \( L \) respectively, or equivalently their scaled versions, \( E^{RMS} \) or \( L^{RMS} \). Although these error calculations can be adapted for use in classification by setting the target classification variable to zeros and ones, this is not recommended. Use of the sum-of-squared and Laplacian error calculations is based on the assumption that the target variable is continuous. In classification applications, the target variable is a discrete random variable with \( C \) possible values, where \( C = \text{number of classes} \).

A multilayer feed-forward neural network for classifying patterns into one of only two categories is referred to as a binary classification network. It has a single output: the estimated probability that the input pattern belongs to one of the two categories. The probably that it belongs to the other category is equal to one minus this probability, i.e.,

\[ P(C_2) = P(\text{not } C_1) = 1 - P(C_1) \]

Binary classification applications are very common. Any problem requiring yes/no classification is a binary classification problem. For example, deciding to sell or buy a stock is a binary classification problem. Deciding to approve a loan application is also a binary classification problem. Deciding whether to approve a new drug or to provide one of two medical treatments are binary classification problems.

For binary classification problems, only a single output is used, \( C = 1 \). This output represents the probability that the training case should be classified as yes. A common choice for the activation function of the output of a binary classification networks is the logistic activation function, which always results in an output in the range 0 to 1, regardless of the perceptron’s potential.
One choice for training a binary classification network is to use sum-of-squared errors with the class value of *yes* patterns coded as a 1 and the *no* classes coded as a 0, i.e.:

\[ t_{ij} = \begin{cases} 
1 & \text{if training pattern } i=\text{yes} \\
0 & \text{if the training pattern } i=\text{no}
\end{cases} \]

However, using either the sum-of-squared or Laplacian errors for training a network with these target values assumes that the noise in the training data are Gaussian. In binary classification, the zeros and ones are not Gaussian. They follow the Bernoulli distribution:

\[
P(t_i = t) = p^t (1 - p)^{1-t}
\]

where \( p \) is equal to the probability that a randomly selected case belongs to the *yes* class.

Modeling the binary classes as Bernoulli observations leads to the cross-entropy error function described by Hopfield (1987) and Bishop (1995):

\[
E^C = - \sum_{i=1}^{N} \left\{ t_i \ln(\hat{t}_i) + (1 - t_i) \ln(1 - \hat{t}_i) \right\}
\]

where \( N \) is the number of training patterns, \( t_i \) is the target value for the \( i \)-th case (either 1 or 0), and \( \hat{t}_i \) is the network’s output for the \( i \)-th case. This is equal to the neural network’s estimate of the probability that the \( i \)-th case should be classified as *yes*.

For situations in which the target variable is a probability in the range \( 0 < t_{ij} < 1 \), the value of the cross-entropy at the networks optimum is equal to:

\[
E^C_{\text{min}} = - \sum_{i=1}^{N} \left\{ t_i \ln(t_i) + (1 - t_i) \ln(1 - t_i) \right\}
\]

Subtracting this from \( E^C \) gives an error term bounded below by zero, i.e., \( E^{CE} \geq 0 \) where:

\[
E^{CE} = E^C - E^C_{\text{min}} = - \sum_{i=1}^{N} \left\{ t_i \ln \left( \frac{\hat{t}_i}{t_i} \right) + (1 - t_i) \ln \left( \frac{1 - \hat{t}_i}{1 - t_i} \right) \right\}
\]

This adjusted cross-entropy is normally reported when training a binary classification network where \( 0 < t_{ij} < 1 \). Otherwise \( E^C \), the non-adjusted cross-entropy error, is used. Small values, values near zero, would indicate that the training resulted in a network with a low error rate and that patterns are being classified correctly most of the time.

**Back-Propagation in Multilayer Feed-Forward Neural Network**

Sometimes a multilayer feed-forward neural network is referred to incorrectly as a back-propagation network. The term back-propagation does not refer to the structure or architecture of a network. Back-propagation refers to the method used during network training. More specifically, back-propagation refers to a simple method for calculating the gradient of the network, that is the first derivative of the weights in the network.
The primary objective of network training is to estimate an appropriate set of network weights based upon a training dataset. Many ways have been researched for estimating these weights, but they all involve minimizing some error function. In forecasting, the most commonly used error function is the sum-of-squared errors:

\[ E = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{C} (t_{ij} - \hat{t}_{ij})^2 \]

Training uses one of several possible optimization methods to minimize this error term. Some of the more common are: steepest descent, quasi-Newton, conjugate gradient, and many various modifications of these optimization routines.

Back-propagation is a method for calculating the first derivative, or gradient, of the error function required by some optimization methods. It is certainly not the only method for estimating the gradient. However, it is the most efficient. In fact, some will argue that the development of this method by Werbos (1974), Parket (1985), and Rumelhart, Hinton and Williams (1986) contributed to the popularity of neural network methods by significantly reducing the network training time and making it possible to train networks consisting of a large number of inputs and perceptrons.

Simply stated, back-propagation is a method for calculating the first derivative of the error function with respect to each network weight. Bishop (1995) derives and describes these calculations for the two most common forecasting error functions, the sum of squared errors and Laplacian error functions. Abe (2001) gives the description for the classification error function, the cross-entropy error function. For all of these error functions, the basic formula for the first derivative of the network weight \( w_{ji} \) at the \( i \)-th perceptron applied to the output from the \( j \)-th perceptron:

\[ \frac{\partial E}{\partial w_{ji}} = \delta_j Z_i, \]

where \( Z_i = g(a_i) \) is the output from the \( i \)-th perceptron after activation, and

\[ \frac{\partial E}{\partial w_{ji}} \]

is the derivative for a single output and a single training pattern. The overall estimate of the first derivative of \( w_{ji} \) is obtained by summing this calculation over all \( N \) training patterns and \( C \) network outputs.

The term back-propagation gets its name from the way the term \( \delta_j \) in the back-propagation formula is calculated:

\[ \delta_j = g'(a_j) \cdot \sum_k w_{kj} \delta_k, \]

where the summation is over all perceptrons that use the activation from the \( j \)-th perceptron, \( g(a_j) \).

The derivative of the activation functions, \( g'(a) \), varies among these functions, see the following table:
**Table 2. Activation Functions and Their Derivatives**

<table>
<thead>
<tr>
<th>Activation Function</th>
<th>( g(a) )</th>
<th>( g'(a) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>( g(a) = a )</td>
<td>( g'(a) = 1 ) (where ( a ) is a constant)</td>
</tr>
<tr>
<td>Logistic</td>
<td>( g(a) = \frac{1}{1+e^{-a}} )</td>
<td>( g'(a) = g(a)(1-g(a)) )</td>
</tr>
<tr>
<td>Hyperbolic-tangent</td>
<td>( g(a) = \tanh(a) )</td>
<td>( g'(a) = \text{sech}^2(a) = 1 - \tanh^2(a) )</td>
</tr>
<tr>
<td>Squash</td>
<td>( g(a) = \frac{a}{1+</td>
<td>a</td>
</tr>
</tbody>
</table>

**Creating a Feed Forward Network**

The following code fragment creates the feed forward neural network shown in the following figure:

![Figure 8. A Three-Layer Feed-Forward Neural Net](image-url)
Notice that this network is more complex than the typical feed-forward network in which all nodes from each layer are connected to every node in the next layer. This network has 6 input nodes, and they are not all connected to every node in the 1st hidden layer.

Note also that the 4 perceptrons in the 1st hidden layer are not connected to every node in the 2nd hidden layer, and the perceptrons in the 2nd hidden layer are not all connected to the two outputs.

```c# // *************************************************************** // EXAMPLE CODE FOR CREATING LINKS AMONG NETWORK NODES // *************************************************************** FeedForwardNetwork network = new FeedForwardNetwork(); network.InputLayer.CreateInputs(6); network.CreateHiddenLayer().CreatePerceptrons(4); network.CreateHiddenLayer().CreatePerceptrons(3); network.OutputLayer.CreatePerceptrons(2); HiddenLayers[] hiddenLayer = network.HiddenLayers; Node[] inputNode = network.InputLayer.Nodes; Node[] layer1Node = hiddenLayer[0].Nodes; Node[] layer2Node = hiddenLayer[1].Nodes; Node[] outputNode = network.OutputLayer.Nodes; // Create links between input nodes and 1st hidden layer network.Link(inputNode[0], layer1Node[0]); network.Link(inputNode[0], layer1Node[1]); network.Link(inputNode[1], layer1Node[0]); network.Link(inputNode[1], layer1Node[1]); network.Link(inputNode[1], layer1Node[3]); network.Link(inputNode[2], layer1Node[1]); network.Link(inputNode[2], layer1Node[2]); network.Link(inputNode[3], layer1Node[3]); network.Link(inputNode[4], layer1Node[3]); network.Link(inputNode[5], layer1Node[3]); // Create links between 1st and 2nd hidden layers network.Link(layer1Node[0], layer2Node[0]); network.Link(layer1Node[0], layer2Node[1]); network.Link(layer1Node[1], layer2Node[0]); network.Link(layer1Node[1], layer2Node[1]); network.Link(layer1Node[1], layer2Node[2]); network.Link(layer1Node[1], layer2Node[2]); network.Link(layer1Node[2], layer2Node[0]); network.Link(layer1Node[2], layer2Node[2]); network.Link(layer1Node[3], layer2Node[1]); network.Link(layer1Node[3], layer2Node[2]); // Create links between 2nd hidden layer and output layer network.Link(layer2Node[0], outputNode[0]); network.Link(layer2Node[1], outputNode[0]); network.Link(layer2Node[1], outputNode[1]); network.Link(layer2Node[2], outputNode[0]); network.Link(layer2Node[2], outputNode[1]); // Create link between input node[0] and output node[0] network.Link(inputNode[0], outputNode[0]); // *************************************************************** By default, the FeedForwardNetwork constructor creates a feed forward network with an empty
input layer, no hidden layers and an empty output layer. Input nodes are created by accessing the empty input layer and creating 6 nodes within it. Two hidden layers are then created within the network using the `FeedForwardNetwork.CreateHiddenLayer().CreatePerceptrons()` method. Four perceptrons are created within the first hidden layer and three within the second. Output perceptrons are created by accessing the empty output layer and creating the `FeedForwardNetwork.OutputLayer.CreatePerceptrons()`.

Links among the input nodes and perceptrons can be created using one of several approaches. If all inputs are connected to every perceptron in the first hidden layer, and if all perceptrons are connected to every perceptron in the following layer, which is a standard architecture for feed forward networks, then a call to the `FeedForwardNetwork.LinkAll()` method can be used to create these links.

However, this example does not use that standard configuration. Some links are missing. In this case, the approach used is to construct individual links using the `FeedForwardNetwork.Link()` method. This requires one call for every link.

An alternate approach is to first create all links and then to remove those that are not needed. The following code illustrates this approach:

```csharp
// **************************************************************
// EXAMPLE CODE FOR REMOVING LINKS AMONG NETWORK NODES
// **************************************************************
FeedForwardNetwork network = new FeedForwardNetwork();
InputNode[] inputNode = network.InputLayer.CreateInputs(6);
Perceptron[] hiddenLayer1 = network.CreateHiddenLayer().CreatePerceptrons(4);
Perceptron[] hiddenLayer2 = network.CreateHiddenLayer().CreatePerceptrons(3);
Perceptron[] outputLayer = network.OutputLayer.CreatePerceptrons(2);
network.LinkAll(); // Creates standard feed forward configuration

// Remove links between input nodes and 1st hidden layer
network.Remove(network.FindLink(inputNode[0], hiddenLayer1[2]));
network.Remove(network.FindLink(inputNode[0], hiddenLayer1[3]));
network.Remove(network.FindLink(inputNode[1], hiddenLayer1[3]));
network.Remove(network.FindLink(inputNode[2], hiddenLayer1[0]));
network.Remove(network.FindLink(inputNode[2], hiddenLayer1[3]));
network.Remove(network.FindLink(inputNode[3], hiddenLayer1[0]));
network.Remove(network.FindLink(inputNode[3], hiddenLayer1[1]));
network.Remove(network.FindLink(inputNode[3], hiddenLayer1[2]));
network.Remove(network.FindLink(inputNode[4], hiddenLayer1[0]));
network.Remove(network.FindLink(inputNode[4], hiddenLayer1[1]));
network.Remove(network.FindLink(inputNode[4], hiddenLayer1[2]));
network.Remove(network.FindLink(inputNode[5], hiddenLayer1[0]));
network.Remove(network.FindLink(inputNode[5], hiddenLayer1[1]));
network.Remove(network.FindLink(inputNode[5], hiddenLayer1[2]));

// Remove links between 1st and 2nd hidden layers
network.Remove(network.FindLink(hiddenLayer1[2], hiddenLayer2[1]));
network.Remove(network.FindLink(hiddenLayer1[3], hiddenLayer2[0]));

// Remove links between 2nd hidden layer and the output layer
network.Remove(network.FindLink(hiddenLayer2[0], outputLayer[1]));

// Add link from input node[0] to output node[0]
network.Link(inputNode[0], outputNode[0]);

// **************************************************************

Neural Nets • 681
In the above fragment, all links are created using the `FeedForwardNetwork.LinkAll()` method. This creates a total of $6 \times 4 + 4 \times 3 + 3 \times 2 = 42$ links, not including the link between the first input node and the first output node. Links that skip layers are not created by the `LinkAll()` method.

Links are then selectively removed starting with the first input node and proceeding to links between the last hidden layer and the output layers. In this case, there are $6 \times 4 = 24$ possible links between the input nodes and first hidden layer. Fourteen of them had to be removed. Between the first hidden layer and second, there are $4 \times 3 = 12$ possible links. Two of them were removed. Between the second hidden layer and output layer there are $3 \times 2 = 6$ possible links, and only one needed to be removed. Finally the skip-layer link between the first input node and first output node is added.

After creating and removing links among layers, the activation function used with each perceptron can be selected. By default, every perceptron in the hidden layers use the logistic activation function and every perceptron in the output layers uses the linear activation function. The following fragment shows how to change the activation function in the hidden layer perceptrons from logistic to hyperbolic-tangent and the output layer from linear to logistic. It also creates a connection directly from the first input node to the output node.

```csharp
// ***************************************************************
// EXAMPLE CODE FOR SETTING NON-DEFAULT ACTIVATION FUNCTIONS
// ***************************************************************

FeedForwardNetwork network = new FeedForwardNetwork();
InputNode[] inputNode = network.InputLayer.CreateInputs(6);
Perceptron[] hiddenLayer1 = network.CreateHiddenLayer().CreatePerceptrons(4);
Perceptron[] hiddenLayer2 = network.CreateHiddenLayer().CreatePerceptrons(3);
Perceptron[] outputLayer = network.OutputLayer.CreatePerceptrons(2);

// Get Network Perceptrons for Setting Their Activation Functions
Perceptron[] perceptrons = network.Perceptrons;

for (int k = 0; k < hiddenLayer1.Length - 1; k++) {
    perceptrons[k].Activation = Imsl.DataMining.Neural.Activation.Tanh;
}

```

### Training

Training

Trainers are used to find the network weights that produce network outputs matching a set of training targets. The training targets together with their associated network inputs are referred to as training patterns. Training patterns can be historical data relating network inputs to its outputs, or they can be developed from expert opinion or theoretical analysis. In the end, each training pattern relates specific network inputs to its real or desired target outputs.
In IMSL C# Numerical Library all trainers implement the `Imsl.DataMining.Neural.ITrainer` interface. The number of training input attributes must equal the number of input nodes, and the number of training outputs, sometimes called training targets, must equal the number of output perceptrons created for the network.

### Single Stage Trainers

`QuasiNewtonTrainer` and `LeastSquaresTrainer` are single stage trainers. They use all available training patterns and a specific optimization method to find optimum network weights. The best set of weights is a set that minimizes the error between the network output and its training targets. The following code fragment illustrates how to use the quasi-Newton method for single stage network training.

```csharp
// ***************************************************************
// EXAMPLE CODE FOR ONE-STAGE TRAINER
// ***************************************************************
double xData[,] = ...
double yData[,] = ...
QuasiNewtonTrainer trainer = new QuasiNewtonTrainer();
trainer.GradientTolerance = 1.0e-7;
trainer.Train(network, xData, yData);
```

In this example, `xData` and `yData` are two-dimensional arrays containing the input attributes and output targets respectively. The number of rows in these arrays is equal to the number of training patterns. The number of columns in `xData` is equal to the number of input attributes, after applying any necessary preprocessing. The number of columns in `yData` is equal to the number of network outputs. The `GradientTolerance` property is one of several optional settings for tailoring the convergence criteria used with the training optimizer.

`LeastSquaresTrainer` is another single stage trainer. There are two principal differences between this trainer and the quasi-Newton trainer. First their optimization algorithms are different. The least squares trainer uses the Levenberg-Marquardt algorithm to optimize the network. As the name implies, the quasi-Newton trainer uses a modified Newton algorithm for optimization. In some applications, depending upon the data and the network architecture, one method may train the network faster than the other.

Another key difference between these single stage trainers is that the least squares trainer only uses one error function, the sum of squared errors. The quasi-Newton trainer, by default, uses the same error function. However, it also has an interface that accepts a user-supplied error function.

### Multistage Trainers

When there are a large number of training patterns, single stage trainers will often take too
long to complete network training. For these applications, a multistage trainer could be used to reduce training time. Multistage trainers provide considerably more flexibility in designing an optimum training scheme. All of these trainers break network training into two stages. Stage II is optional. That is, a multistage trainer can be requested to only conduct Stage I training, or it can be requested to conduct both Stage I and II training.

The main difference between Stage I and II training is that Stage I training is conducted multiple times using randomly selected subsets of all available training patterns. Each training session is referred to as an epoch. Although each epoch uses a different set of randomly selected training patterns, the number of patterns is the same for every epoch. Typically, because they are using different data, the solutions vary among epochs.

Stage II training is conducted following the Stage I training using the best set of weights obtained during Stage I. This ensures that the weights developed during Stage II training will always be as good as or better than those determined during Stage I training. The entire set of original training patterns is used during Stage II training, and only one training session is completed.

There is no requirement to use the same trainer for both stages, although there is nothing wrong with that approach. The least squares trainer might be used for Stage I training and the quasi-Newton trainer might be used for Stage II training. In addition, the optimization settings for each trainer can be different. The multistage trainer is implemented using the *EpochTrainer* class.

The following code fragment illustrates the use of the epoch multistage trainer:

```csharp
// **************************************************************
// EXAMPLE CODE FOR MULTISTAGE EPOCH TRAINER
// **************************************************************
    double xData[,] = ... double yData[,] = ...    
    QuasiNewtonTrainer stageITrainer = new QuasiNewtonTrainer();
    LeastSquaresTrainer stageIITrainer = new LeastSquaresTrainer();
    EpochTrainer trainer = new EpochTrainer(stageITrainer, stageIITrainer);
    trainer.NumberOfEpochs = 20;
    trainer.EpochSize = 3000;
.
.
// **************************************************************
```

In this example, a quasi-Newton trainer is selected for the Stage I trainer, and the least squares trainer is used for Stage II. Stage I will consists of 20 training epochs. The training of each epoch uses 3,000 randomly selected training patterns with the quasi-Newton trainer. The epoch with the smallest training error supplies the starting values for the Stage II trainer.

**Data Preprocessing**

Data preprocessing, or filtering, is the term used to describe the process of scaling or transforming input attributes into numerical values suitable for network training. In general it
is important to scale all input attributes to a common range, either \([0, 1]\) or \([-1, 1]\). The algorithm used for obtaining values for the network weights assumes that the inputs are scaled to one of these ranges. If some network inputs have values that cover a much broader range, then the initial weights can be far from optimum causing network training to fail or take an excessively long time.

Network input data are classified into three general categories: continuous, ordinal and nominal. IMSL C# Numerical Library provides methods for preprocessing all three data types. Continuous data are scaled using the ScaleFilter class. In addition, lagged versions of continuous time series data can be created using the TimeSeriesFilter or TimeSeriesClassFilter class.

Categorical data, such as color or preference ratings, are either ordinal and nominal data. UnsupervisedOrdinalFilter and UnsupervisedNominalFilter are provided to preprocess ordinal and nominal data respectively. UnsupervisedOrdinalFilter transforms ordinal data into values between 0 and 1, which allows them to be treated as continuous data.

Nominal data, on the other hand, can be transformed using several methods. UnsupervisedNominalFilter converts a single nominal variable with \(m\) classes into \(m\) columns containing the values 0 and 1. This is referred to as binary encoding of nominal classification information.

The following code fragment illustrates the use of some of these preprocessing methods:

```csharp
// ***************************************************************
// EXAMPLE CODE FOR PREPROCESSING NOMINAL AND CONTINUOUS DATA
// ***************************************************************

double[,] yData = {...};
int[] nominalVariable={......};
int nClasses = 3;

// Create a nominal filter for binary encoding of a nominal variable
// that has 3 categorical values
UnsupervisedNominalFilter nominalFilter = new UnsupervisedNominalFilter(nClasses);
int[,] binaryColumns = nominalFilter.Encode(nominalVariable);

// Create a scale filter for scaling continuous data in a range of [0,1]
ScaleFilter scaleFilter = new ScaleFilter(ScaleFilter.ScalingMethod.Bounded);
// Apply the scale filter to two continuous variables, x1 and x2
scaleFilter.SetBounds(-200,1000,0,1); // Original values [-200, 1000]
scaleFilter.Encode(x1);
scaleFilter.SetBounds(0,5000,0,1); // Original values [0, 5000]
scaleFilter.Encode(x2);

// Load the encoded columns into xData
int n = nominalVariable.Length;
double[,] xData = new double[n, 3+3];
for(int i=0; i < n; i++)
    { xData[i,0] = x1[i];
      xData[i,1] = x2[i];
      for(int j=0; j < nClasses; j++)
          xData[i,j+2] = binaryColumns[i,j];
    }
```

Neural Nets • 685
In the above example, one nominal variable consisting of values representing 3 different classes, or categories, is encoded into 3 binary columns using `UnsupervisedNominalFilter` class. Two continuous variables are scaled using the `ScaleFilter` class, and these five columns are then loaded into xData in preparation for network training.

**Serialization**

Neural network training can require a substantial amount of time, so it is often desirable to save a trained network for later use in forecasting. Serialization can be used to save the results of network training.

When an object is serialized, its state is saved. However, the code implementing the class (the class file) is not saved with the serialized file. Hence when the object is deserialized, the code that created the serialized object should be in the classpath. Otherwise deserialization will fail.

For an object to be serialized, the class must use the `Serializable` attribute. The following code fragment serializes key network and training information into four files. One contains the network weights, another contains the training statistics, and two additional files contain the training patterns. This is done using a `write(Object, String)` method that takes a file name and writes the serialized object to that file.

```csharp
// ***************************************************************
// EXAMPLE CODE FOR SAVING TRAINED NETWORK USING SERIALIZATION
// ***************************************************************
using System.Runtime.Serialization;

using System.IO;

// ***************************************************************
// SAVE A TRAINED NETWORK BY SAVING THE SERIALIZED NETWORK OBJECTS
// ***************************************************************
// Saving network weights and structural information
write(network, "MyNetwork.ser");
// Saving training information available from computeStatistics()
write(trainer, "MyNetworkTrainer.ser");
// Saving xData training targets
write(xData, "MyNetworkxData.ser");
// Saving yData training targets
write(yData, "MyNetworkyData.ser");

// WRITE SERIALIZED NETWORK TO A FILE
// ***************************************************************
static public void write(System.Object obj, System.String filename)
{
System.IO.FileStream fos = new System.IO.FileStream(filename, System.IO.FileMode.Create);
IFormatter oos = new BinaryFormatter();
Notice that not only is the network object serialized and saved, the trainer and training patterns, xData and yData, are also saved. This was only done to allow someone to calculate the additional network statistics. If these are not needed, then these training patterns need not be saved. However, for forecasting, it is essential to remember the specific order and nature of the network inputs used during training. This information is not saved in the network serialized file.

When an object is deserialized, the object is reconstructed using the saved serialization file. The following code deserializes the previously saved network information.

```csharp
// ***************************************************************
// EXAMPLE CODE FOR READING TRAINED NETWORK FROM SERIALIZED FILES
using System.Runtime.Serialization;

// ***************************************************************
// READ THE TRAINED NETWORK FROM THE SERIALIZED NETWORK OBJECT
Network network = (Network)read("MyNetwork.ser");
// READ THE Serialized XDATA[,] AND YDATA[,] ARRAYS OF TRAINING PATTERNS.
xData = (double[,] )read("MyNetworkxData.ser");
yData = (double[,] )read("MyNetworkyData.ser");
// READ THE Serialized TRAINER OBJECT
Trainer trainer = (ITrainer)read("MyNetworkTrainer.ser");
// ***************************************************************
// DISPLAY TRAINING STATISTICS

double stats[] = network.computeStatistics(xData, yData);

// ***************************************************************
// READ SERIALIZED NETWORK FROM A FILE
static public System.Object read(System.String filename)
{
    System.IO.FileStream fis = new System.IO.FileStream(filename, System.IO.FileMode.Open, System.IO.FileAccess.Read);
    IFormatter ois = new BinaryFormatter();
    System.Object obj = (System.Object) ois.Deserialize(fis);
    fis.Close();
    return obj;
}
// ***************************************************************

Neural Nets • 687
FeedForwardNetwork Class

A representation of a feed forward neural network.

public class Imsl.DataMining.Neural.FeedForwardNetwork : Network

Properties

HiddenLayers
  virtual public Imsl.DataMining.Neural.HiddenLayer[] HiddenLayers {get; }
  The HiddenLayers in this Imsl.DataMining.Neural.Network (p. 762). An array of
  HiddenLayers in this Network.

InputLayer
  override public Imsl.DataMining.Neural.InputLayer InputLayer {get; }
  The InputLayer in this Imsl.DataMining.Neural.Network (p. 762). The neural network
  InputLayer.

Links
  override public Imsl.DataMining.Neural.Link[] Links {get; }
  All the Links in this Imsl.DataMining.Neural.Network (p. 762). An array of Links
  containing all of the Links in this Network.

NumberOfInputs
  override public int NumberOfInputs {get; }
  The number of inputs to the Imsl.DataMining.Neural.Network (p. 762). An int
  containing the number of inputs to the Network.

NumberOfLinks
  override public int NumberOfLinks {get; }
  The number of
  (p. 690)s in the Imsl.DataMining.Neural.Network (p. 762). An int which contains the
  number of Links in the Network.

NumberOfOutputs
  override public int NumberOfOutputs {get; }
  The number of outputs from the Imsl.DataMining.Neural.Network (p. 762). An int
  containing the number of outputs from the Network.

NumberOfWeights
  override public int NumberOfWeights {get; }

OutputLayer

    override public Imsl.DataMining.Neural.OutputLayer OutputLayer {get; }

The neural network OutputLayer. Contains the neural network OutputLayer.

Perceptrons

    override public Imsl.DataMining.Neural.Perceptron[] Perceptrons {get; }


Weights

    override public double[] Weights {get; set; }


The array contains the Weights for each Link followed by the Perceptron Imsl.DataMining.Neural.Perceptron.Bias (p. 706) values. The Link Weights are the order in which the Links were created. The Weight values are first, followed by the Bias values in the Imsl.DataMining.Neural.HiddenLayer (p. 701) and then the Bias values in the Imsl.DataMining.Neural.FeedForwardNetwork.OutputLayer (p. 689), and then by the order in which the Imsl.DataMining.Neural.Perceptron (p. 706)s were created.

Constructor

FeedForwardNetwork

    public FeedForwardNetwork()

    Creates a new instance of FeedForwardNetwork.

Methods

ComputeStatistics

    virtual public double[] ComputeStatistics(double[,] xData, double[,] yData)

    Computes error statistics.

CreateHiddenLayer

    override public Imsl.DataMining.Neural.HiddenLayer CreateHiddenLayer()

    Creates a HiddenLayer.

    Returns — A HiddenLayer object which specifies a neural network hidden layer.
FindLink

virtual public Imsl.DataMining.Neural.Link
Returns the Link between two Nodes.

from  The origination Node.

to    The destination Node.

Returns — A Link between the two Nodes, or null if no such Link exists.

FindLinks

virtual public Imsl.DataMining.Neural.Link[]
FindLinks(Imsl.DataMining.Neural.Node to)
Returns all of the Links to a given Node.

to     A Node whose Links are to be determined.

Returns — An array of Links containing all of the Links to the given Node.

Forecast

override public double[] Forecast(double[] x)
Computes a forecast using the Imsl.DataMining.Neural.Network (p. 762).

x     A double array of values to which the Imsl.DataMining.Neural.Node (p. 704)s in the
Imsl.DataMining.Neural.FeedForwardNetwork.InputLayer (p. 688) are to be set.

Returns — A double array containing the values of the Nodes in the

GetForecastGradient

override public double[] GetForecastGradient(double[] xData)
Returns the gradient with respect to the Imsl.DataMining.Neural.Link.Weight (p. 709)s.
The i-th entry in the resulting array contains \( dN(xData, weights)/d weights[i] \).

xData   A double array which specifies the input values at which the gradient is to be
evaluated.

Returns — A double array containing the gradient values.

Link

Establishes a Link between two Nodes with a specified
Imsl.DataMining.Neural.Link.Weight (p. 709).

from  The origination Node.

to    The destination Node.

weight  A double which specifies the Weight to be given the Link.

Returns — A Link between the two Nodes.
**Link**

```csharp
Establishes a Link between two Nodes.
Any existing Link between these Nodes is removed.

*from* The origination Node.
*to* The destination Node.

Returns — A Link between the two Nodes.
```

**LinkAll**

```csharp
virtual public void LinkAll()
```

```csharp
virtual public void LinkAll(Imsl.DataMining.Neural.Layer from, Imsl.DataMining.Neural.Layer to)
Links all of the Imsl.DataMining.Neural.Node (p. 704) s in one Layer to all of the Nodes in another Layer.

*from* The origination Layer.
*to* The destination Layer.
```

**Remove**

```csharp
virtual public void Remove(Imsl.DataMining.Neural.Link link)

*link* The Link deleted from the Network.
```

**ValidateLink**

```csharp

In a feed forward network a link must be from a node in one layer to a node in a later layer. Intermediate layers can be skipped, but a link cannot go backward.

*from* The origination Node.
*to* The destination Node.

Imsl.Messages.ThrowArgumentException(System.String, System.String, System.Object[]) is thrown if the Link is not valid
Description

A **Network** contains an `Imsl.DataMining.Neural.FeedForwardNetwork.InputLayer` (p. 688), an `Imsl.DataMining.Neural.FeedForwardNetwork.OutputLayer` (p. 689) and zero or more `Imsl.DataMining.Neural.HiddenLayer` (p. 701)s. The null `InputLayer` and `OutputLayer` are automatically created by the `Network` constructor. The `Imsl.DataMining.Neural.InputNode` (p. 705)s are added using the `FeedForwardNetwork.InputLayer.CreateInputs(nInputs)` method. Output `Imsl.DataMining.Neural.Perceptron` (p. 706)s are added using the `FeedForwardNetwork.OutputLayer.CreatePerceptrons(nOutputs)`, and `HiddenLayers` can be created using the `FeedForwardNetwork.CreateHiddenLayer() . CreatePerceptrons(nPerceptrons)` method.


Each `Link` has a `Imsl.DataMining.Neural.Link.Weight` (p. 709) and `gradient` value. Each `Perceptron` node has a `Imsl.DataMining.Neural.Perceptron.Bias` (p. 706) value. When the `Network` is trained, the `Weight` and `Bias` values are used as initial guesses. After the `Network` is trained the `Weight`, `gradient` and `Bias` values are set to the values computed by the training.

A feed forward network is a network in which links are only allowed from one layer to a following layer.

**Example: FeedForwardNetwork**

This example trains a 2-layer network using 100 training patterns from one nominal and one continuous input attribute. The nominal attribute has three classifications which are encoded using binary encoding. This results in three binary network input columns. The continuous input attribute is scaled to fall in the interval [0,1].

The network training targets were generated using the relationship:

\[ y = 10 \times X_1 + 20 \times X_2 + 30 \times X_3 + 2.0 \times X_4, \]

where

- \( X_1 \) to \( X_3 \) are the three binary columns, corresponding to categories 1-3 of the nominal attribute,
- \( X_4 \) is the scaled continuous attribute.

The structure of the network consists of four input nodes and two layers, with three perceptrons in the hidden layer and one in the output layer. The following figure illustrates this structure:
There are a total of 19 weights in this network. The activations functions are all linear. Since
the target output is a linear function of the input attributes, linear activation functions
guarantee that the network forecasts will exactly match their targets. Of course, this same
result could have been obtained using linear multiple regression. Training is conducted using
the quasi-newton trainer.

using System;
using Imsl.DataMining.Neural;
using System.Runtime.Serialization;

//*****************************************************************************
// Two Layer Feed-Forward Network with 4 inputs: 1 nominal with 3 categories,
// encoded using binary encoding, 1 continuous input attribute, and 1 output
// target (continuous).
// There is a perfect linear relationship between the input and output
// variables:
// MODEL: Y = 10*X1+20*X2+30*X3+2*X4

Neural Nets FeedForwardNetwork Class • 693
Variables X1-X3 are the binary encoded nominal variable and X4 is the continuous variable.

//****************************************************************************
public class FeedForwardNetworkEx1 //: System.Runtime.Serialization.ISerializable
{
    // Network Settings
    private static int nObs = 100; // number of training patterns
    private static int nInputs = 4; // four inputs
    private static int nCategorical = 3; // three categorical attributes
    private static int nOutputs = 1; // one continuous output
    private static int nPerceptrons = 3; // perceptrons in hidden layer
    private static IActivation hiddenLayerActivation;
    private static IActivation outputLayerActivation;
    private static System.String errorMsg = "";
    // Error Status Messages for the Least Squares Trainer
    private static System.String errorMsg0 = "--> Least Squares Training Completed Successfully";
    private static System.String errorMsg1 = "--> Scaled step tolerance was satisfied. The current solution \n" + "may be an... or the algorithm is making\n" + "slow progress and is not near a solution, or the Step Tolerance
" + "is too big";
    private static System.String errorMsg2 = "--> Scaled actual and predicted reductions in the function are\n" + "less than or equal to the relative function convergence
" + "tolerance RelativeTolerance";
    private static System.String errorMsg3 = "--> Iterates appear to be converging to a noncritical point.\n" + "Incorrect gradient information, a discontinuous function,
" + "or stopping tolerances being too tight may be the cause.";
    private static System.String errorMsg4 = "--> Five consecutive steps with the maximum stepsize have\n" + "been taken. It is possible that the optimal solution is unbounded below, or has
" + "a finite asymptote in some direction, or the maximum stepsize
" + "is too small.";
    private static System.String errorMsg5 = "--> Too many iterations required";

    // categoricalAtt[]: A 2D matrix of values for the categorical training
    // attribute. In this example, the single categorical
    // attribute has 3 categories that are encoded using
    // binary encoding for input into the network.
    // {1,0,0} = category 1, {0,1,0} = category 2, and
    // {0,0,1} = category 3.
    private static double[,] categoricalAtt =
    {{1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0},
    {1, 0, 0}, {1, 0, 0}, {0, 1, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0},
    {1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0},
    {1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0},
    {1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0},
    {0, 1, 0}, {0, 1, 0}, {0, 1, 0}, {0, 1, 0}, {0, 1, 0}, {0, 1, 0},
    {0, 1, 0}, {0, 1, 0}, {0, 1, 0}, {0, 1, 0}, {0, 1, 0}, {0, 1, 0},
    {0, 1, 0}, {0, 1, 0}, {0, 1, 0}, {0, 1, 0}, {0, 1, 0}, {0, 1, 0},
    {0, 1, 0}, {0, 1, 0}, {0, 1, 0}, {0, 1, 0}, {0, 1, 0}, {0, 1, 0},
    {0, 0, 1}, {0, 0, 1}, {0, 0, 1}, {0, 0, 1}, {0, 0, 1}, {0, 0, 1},
    {0, 0, 1}, {0, 0, 1}, {0, 0, 1}, {0, 0, 1}, {0, 0, 1}, {0, 0, 1},
    {0, 0, 1}, {0, 0, 1}, {0, 0, 1}, {0, 0, 1}, {0, 0, 1}, {0, 0, 1},
    {0, 0, 1}, {0, 0, 1}, {0, 0, 1}, {0, 0, 1}, {0, 0, 1}, {0, 0, 1}};

    // contAtt[]: A matrix of values for the continuous training attribute
    private static double[] contAtt = new double[]{4.007054658, 7.10028447, 4.740350984, 5.714553211, 6.205437459, ...};

    // outs[]: A 2D matrix containing the training outputs for this network
}
In this case there is an exact linear relationship between these outputs and the inputs: \( \text{outs} = 10 \times X_1 + 20 \times X_2 + 30 \times X_3 + 2 \times X_4 \), where \( X_1 - X_3 \) are the categorical variables and \( X_4 = \text{contAtt} \).

```java
private static double[] outs = new double[] {18.01410932, 24.20056894, 19.48070197, 21.42910642, 22.41087492, ... 32.4350953, 37.81775874, 38.50662776, 48.62379392, 37.62390767, 41.56943258, 36.8289729, 48.69082603, 32.04810755};
```

```
// MAIN
// **********************************************************************
public static void Main(System.String[] args)
{
    double[] weight; // network weights
double[] gradient; // network gradient after training
double[,] xData; // Input Attributes for Trainer
double[,] yData; // Output Attributes for Trainer
int i, j; // array indices
int nWeights = 0; // Number of weights obtained from network
System.String networkFileName = "FeedForwardNetworkEx1.ser";
System.String trainerFileName = "FeedForwardTrainerEx1.ser";
System.String xDataFileName = "FeedForwardxDataEx1.ser";
System.String yDataFileName = "FeedForwardyDataEx1.ser";
System.String trainLogName = "FeedForwardTraining.log";
// **********************************************************************
// PREPROCESS TRAINING PATTERNS
// **********************************************************************
System.Console.Out.WriteLine("-->
```
network.OutputLayer.CreatePerceptrons(nOutputs);
// link all inputs and perceptrons to all perceptrons in the next layer
network.LinkAll();
// Get Network Perceptrons for Setting Their Activation Functions
Perceptron[] perceptrons = network.Perceptrons;
// Set all perceptrons to linear activation
for (i = 0; i < perceptrons.Length - 1; i++)
{
    perceptrons[i].Activation = hiddenLayerActivation;
}
perceptrons[perceptrons.Length - 1].Activation = outputLayerActivation;
System.Console.Out.WriteLine("-->
Feed Forward Network Created with 2 Layers");
// ***************************************************************
// TRAIN NETWORK USING QUASI-NEWTON TRAINER
// ***************************************************************
System.Console.Out.WriteLine("-->
Training Network using Quasi-Newton Trainer");
// Create Trainer
QuasiNewtonTrainer trainer = new QuasiNewtonTrainer();
// Set Training Parameters
trainer.MaximumTrainingIterations = 1000;
// If tracing is requested setup training logger
// if (trace)
// {
// try
// {
//     Handler handler = new FileHandler(trainLogName);
//     Logger logger = Logger.getLogger("Ims1.DataMining.Neural");
//     logger.setLevel(Level.FINEST);
//     logger.addHandler(handler);
//     handler.setFormatter(QuasiNewtonTrainer.Formatter);
Training Log Created in " + trainLogName);
// }
// catch (System.Exception e)
// {
Cannot Create Training Log.");
// }
// }
// Train Network
trainer.Train(network, xData, yData);
// Check Training Error Status
switch (trainer.ErrorStatus)
{
    case 0: errorMsg = errorMsg0;
            break;
    case 1: errorMsg = errorMsg1;
            break;
    case 2: errorMsg = errorMsg2;
            break;
    case 3: errorMsg = errorMsg3;
            break;
    case 4: errorMsg = errorMsg4;
break;

case 5:  errorMsg = errorMsg5;
        break;

default:  errorMsg = errorMsg0;
        break;
}

System.Console.Out.WriteLine(errorMsg);

// **********************************************************************
// DISPLAY TRAINING STATISTICS
// **********************************************************************

double[] stats = network.ComputeStatistics(xData, yData);

// Display Network Errors
System.Console.Out.WriteLine("**********************");
System.Console.Out.WriteLine("---> SSE: "+(float) stats[0]);
System.Console.Out.WriteLine("---> RMS: "+(float) stats[1]);
System.Console.Out.WriteLine("---> Largest Absolute Residual: "+(float) stats[4]);
System.Console.Out.WriteLine("**********************");

//**********************************************************************
// OBTAIN AND DISPLAY NETWORK WEIGHTS AND GRADIENTS
//**********************************************************************


weight = network.Weights;

nWeights = network.NumberOfWeights;

for (i = 0; i < nWeights; i++) {
    System.Console.Out.WriteLine("w["+ i + "]="+(float) weight[i] + " g["+ i + "]=" + (float) gradient[i]);
}

System.Console.Out.WriteLine("**********************");

System.Console.Out.WriteLine("--> Saving Trained Network into " + networkFileName);
write(network, networkFileName);
System.Console.Out.WriteLine("--> Saving xData into " + xDataFileName);
write(xData, xDataFileName);
System.Console.Out.WriteLine("--> Saving yData into " + yDataFileName);
write(yData, yDataFileName);
System.Console.Out.WriteLine("--> Saving Network Trainer into " + trainerFileName);
write(trainer, trainerFileName);
}

//****************************************************************************
// WRITE SERIALIZED NETWORK TO A FILE
//****************************************************************************
static public void write(System.Object obj, System.String filename)
{
    System.IO.FileStream fos = new System.IO.FileStream(filename, System.IO.FileMode.Create);
    IFormatter oos = new BinaryFormatter();
    oos.Serialize(fos, obj);
    fos.Close();
}

static FeedForwardNetworkEx1()
{
    hiddenLayerActivation = Imsl.DataMining.Neural.Activation.Linear;
    outputLayerActivation = Imsl.DataMining.Neural.Activation.Linear;
}

Output

--> Starting Preprocessing of Training Patterns
--> Creating Feed Forward Network Object
--> Feed Forward Network Created with 2 Layers
--> Training Network using Quasi-Newton Trainer
--> Least Squares Training Completed Successfully
******************************************************************************
--> SSE: 1.013444E-15
--> RMS: 2.007463E-19
--> Laplacian Error: 3.005804E-07
--> Scaled Laplacian Error: 3.535235E-10
--> Largest Absolute Residual: 2.784275E-08
******************************************************************************

--> Getting Network Weights and Gradients

--> Network Weights and Gradients:
******************************************************************************
w[0]=-1.491785 g[0]=-2.611079E-08
w[1]=-1.491785 g[1]=-2.611079E-08
w[8]=4.725622 g[8]=-5.273856E-08
w[12]=1.072258 g[12]=-1.690978E-07

698 • FeedForwardNetwork Class  IMSL C# Numerical Library
Layer Class

The base class for Layers in a neural network.

public class Imsl.DataMining.Neural.Layer

Properties

Index

virtual public int Index {get; set; }
The Index of this Layer.

Nodes

virtual public Imsl.DataMining.Neural.Node[] Nodes {get; }
A list of the Nodes in this Layer. An array containing the Nodes associated with this Layer.

Constructor

Layer

protected internal Layer(Imsl.DataMining.Neural.FeedForwardNetwork network)
Constructs a Layer.

network The FeedForwardNetwork to which this Layer is associated.

Method

AddNode

virtual protected internal void AddNode(Imsl.DataMining.Neural.Node node)
Associates a Imsl.DataMining.Neural.Perceptron (p. 706) with this Layer.
A Node to associate with this Layer.

See Also


InputLayer Class

Input layer in a neural network.

public class Imsl.DataMining.Neural.InputLayer : Layer

Properties

Index

    virtual public int Index {get; set; }
    The Index of this Layer.

Nodes

    override public Imsl.DataMining.Neural.Node[] Nodes {get; }

Methods

AddNode

    virtual protected internal void AddNode(Imsl.DataMining.Neural.Node node)
    Associates a Imsl.DataMining.Neural.Perceptron (p. 706) with this Layer.

CreateInput

    virtual public Imsl.DataMining.Neural.InputNode CreateInput()
    Creates an InputNode in the InputLayer of the neural network.

CreateInputs

    virtual public Imsl.DataMining.Neural.InputNode[] CreateInputs(int n)
    Creates a number of InputNodes in this Imsl.DataMining.Neural.Layer (p. 699) of the neural network.

    n  An int which specifies the number of InputNodes to be created in this Layer.
Returns — An array containing the created InputNodes.

**Description**

An InputLayer is automatically created by Network.

**See Also**

Imsl.DataMining.Neural.Network (p. 762)

---

**HiddenLayer Class**

Hidden layer in a neural network. This is created by a factory method in Imsl.DataMining.Neural.Network (p. 762).

```csharp
public class Imsl.DataMining.Neural.HiddenLayer : Layer
```

**Properties**

- **Index**
  ```csharp
  virtual public int Index {get; set; }
  The Index of this Layer.
  ```

- **Nodes**
  ```csharp
  virtual public Imsl.DataMining.Neural.Node[] Nodes {get; }
  A list of the Nodes in this Layer.
  ```

**Methods**

- **AddNode**
  ```csharp
  virtual protected internal void AddNode(Imsl.DataMining.Neural.Node node)
  Associates a Imsl.DataMining.Neural.Perceptron (p. 706) with this Layer.
  ```

- **CreatePerceptron**
  ```csharp
  virtual public Imsl.DataMining.Neural.Perceptron CreatePerceptron()
  Creates a Perceptron in this Imsl.DataMining.Neural.Layer (p. 699) of the neural network.
  The created Perceptron uses the logistic activation function and has an initial Imsl.DataMining.Neural.Perceptron.Bias (p. 706) value of zero.
  ```
CreatePerceptron

virtual public Imsl.DataMining.Neural.Perceptron CreatePerceptron(Imsl.DataMining.Neural.IActivation activation, double bias)


**activation**  The IActivation object which specifies the activation function to be used.

**bias**  A double which specifies the initial value for the Bias.

CreatePerceptrons

virtual public Imsl.DataMining.Neural.Perceptron[] CreatePerceptrons(int n, Imsl.DataMining.Neural.IActivation activation, double bias)

Creates a number of Perceptrons in this Imsl.DataMining.Neural.Layer (p. 699) with the specified Imsl.DataMining.Neural.Perceptron.Bias (p. 706).

**n**  An int which specifies the number of Perceptrons to be created.

**activation**  The IActivation object which specifies the action function to be used.

**bias**  A double containing the initial value to be applied as the Bias values for the Perceptrons.

Returns — An array containing the created Perceptrons.

CreatePerceptrons

virtual public Imsl.DataMining.Neural.Perceptron[] CreatePerceptrons(int n)

Creates a number of Perceptrons in this Imsl.DataMining.Neural.Layer (p. 699) of the neural network.

The created Perceptrons use the logistic activation function and have an initial Imsl.DataMining.Neural.Perceptron.Bias (p. 706) value of zero.

**n**  An int which specifies the number of Perceptrons to be created.

Returns — An array containing the created Perceptrons.

**See Also**

Imsl.DataMining.Neural.Network.CreateHiddenLayer (p. 763)

---

**OutputLayer Class**

Output layer in a neural network.

public class Imsl.DataMining.Neural.OutputLayer : Layer
Properties

Index
virtual public int Index {get; set;}  
The Index of this Layer.

Nodes
override public Imsl.DataMining.Neural.Node[] Nodes {get;}  
The Imsl.DataMining.Neural.Perceptron (p. 706)s in the 
Imsl.DataMining.Neural.OutputLayer (p. 702). An 
Imsl.DataMining.Neural.OutputPerceptron (p. 707) array containing the Nodes in the 
OutputLayer. 
This method overrides the method in Imsl.DataMining.Neural.Layer (p. 699) to return the Perceptrons in an OutputPerceptron array.

Methods

AddNode
virtual protected internal void AddNode(Imsl.DataMining.Neural.Node node)  
Associates a Imsl.DataMining.Neural.Perceptron (p. 706) with this Layer.

CreatePerceptron
virtual public Imsl.DataMining.Neural.Perceptron CreatePerceptron()  
Creates a Perceptron in this Imsl.DataMining.Neural.Layer (p. 699) of the neural 
network. By default, the created Perceptron uses the linear activation function and has 

CreatePerceptron
virtual public Imsl.DataMining.Neural.Perceptron CreatePerceptron(Imsl.DataMining.Neural.IActivation activation, double bias)  
Creates a Perceptron in this Imsl.DataMining.Neural.Layer (p. 699) with a specified 

activation  The Activation object which specifies the action function to be used. 
bias  A double which specifies the initial value for the Bias for this Perceptron.

CreatePerceptrons
virtual public Imsl.DataMining.Neural.Perceptron[] CreatePerceptrons(int n, 
Imsl.DataMining.Neural.IActivation activation, double bias)  
Creates a number of Perceptrons in this Imsl.DataMining.Neural.Layer (p. 699) with 
n  An int which specifies the number of Perceptrons to be created. 
activation  The Activation object which indicates the action function to be used.
bias  A double which specifies the initial Bias for the Perceptrons.

Returns — An array containing the created Perceptrons.

CreatePerceptrons

    virtual public Imsl.DataMining.Neural.Perceptron[] CreatePerceptrons(int n)

Creates a number of Perceptrons in this Imsl.DataMining.Neural.Layer (p. 699) of the neural network. By default, they will use linear activation and a zero initial Imsl.DataMining.Neural.Perceptron.Bias (p. 706).

    n    An int which specifies the number of Perceptrons to be created in this Layer.

Returns — An array containing the created Perceptrons.

Description

An empty OutputLayer is automatically created by Imsl.DataMining.Neural.FeedForwardNetwork (p. 688).

See Also

Imsl.DataMining.Neural.Network (p. 762)

Node Class

A Node in a neural network.

    public class Imsl.DataMining.Neural.Node

Property

    Layer

        virtual public Imsl.DataMining.Neural.Layer Layer {get; }

        The Layer in which this Node exists. The Layer associated with this Node.

Methods

    GetValue

        virtual public double GetValue()

        Returns the value of this Node.

        Returns — A double which contains the value of the Node.
SetValue
   virtual public void SetValue(double node)
   Sets the value of this Node.

   node A double which specifies a value for the Node.

Description

Node is an abstract class that serves as the base class for the concrete classes InputNode and Perceptron.

See Also


---

InputNode Class


public class Imsl.DataMining.Neural.InputNode : Node

Property

Layer
   virtual public Imsl.DataMining.Neural.Layer Layer {get; }
   The Layer in which this Node exists.

Methods

GetValue
   override public double GetValue()
   Returns the value of this Imsl.DataMining.Neural.Node (p. 704).
   Returns — A double which contains the value of this InputNode.

SetValue
   override public void SetValue(double node)
   Sets the value of this Imsl.DataMining.Neural.Node (p. 704).
   node A double which specifies the new value of this InputNode.
Description

InputNodes are not created directly. Instead factory methods in InputLayer are used to create InputNodes within the InputLayer. For example, Imsl.DataMining.Neural.InputLayer.CreateInput (p. 700) creates a single InputNode.

Perceptron Class

A Perceptron node in a neural network.

```csharp
public class Imsl.DataMining.Neural.Perceptron : Node
```

Properties

Activation

```csharp
virtual public Imsl.DataMining.Neural.IActivation Activation {get; set; }
```

The activation function. An Activation object which represents the activation $g$ used by this Perceptron.

Bias

```csharp
virtual public double Bias {get; set; }
```

The Bias for this perceptron. A double representing the Bias for this Perceptron.

The Bias has a default value of 0.

Layer

```csharp
virtual public Imsl.DataMining.Neural.Layer Layer {get; }
```

The Layer in which this Node exists.

Methods

GetValue

```csharp
virtual public double GetValue()
```

Returns the value of this Node.

SetValue

```csharp
virtual public void SetValue(double node)
```

Sets the value of this Node.
Description

Perceptrons are created by factory methods in a Imsl.DataMining.Neural.Layer (p. 699). Each Perceptron has an Imsl.DataMining.Neural.Perceptron.Activation (p. 706) function \( g \) and an Imsl.DataMining.Neural.Perceptron.Bias (p. 706) \( (\mu) \). The value of a Perceptron is given by \( g(\sum_i w_i X_i + \mu) \), where \( X_i \)s are the values of nodes input to this Perceptron with Imsl.DataMining.Neural.Link.Weight (p. 709) \( (w_i) \).

Imsl.DataMining.Neural.Network (p. 762) training will use existing Bias values for the starting values for the trainer. Upon completion of Network training, the Bias values are set to the values computed by the trainer.

OutputPerceptron Class

A Perceptron in the output layer.

public class Imsl.DataMining.Neural.OutputPerceptron : Perceptron

Properties

Activation

virtual public Imsl.DataMining.Neural.IActivation Activation {get; set; }

The activation function.

Bias

virtual public double Bias {get; set; }

The Bias for this perceptron.

Layer

virtual public Imsl.DataMining.Neural.Layer Layer {get; }

The Layer in which this Node exists.

Methods

GetValue

override public double GetValue()

Returns the value of the output perceptron determined using the current Imsl.DataMining.Neural.Network (p. 762) state and inputs.

Returns — A double value of the output perceptron determined using the current Network state and inputs.

SetValue

virtual public void SetValue(double node)
Sets the value of this Node.

Description

OutputPerceptrons are created by factory methods in OutputLayer.

OutputPerceptrons are not created directly. Instead factory methods in OutputLayer are used to create OutputPerceptrons within the OutputLayer. For example, OutputLayer.createPerceptron() creates a single OutputPerceptron.

See Also

Imsl.DataMining.Neural.OutputLayer (p. 702)

IActivation Interface

Interface implemented by perceptron activation functions.

public interface Imsl.DataMining.Neural.IActivation

Methods

Derivative

abstract public double Derivative(double x, double y)
Returns the value of the derivative of the activation function.

y is not mathematically required, but can sometimes be used to more quickly compute the derivative.

x A double which specifies the point at which the activation function is to be evaluated.

y A double which specifies y = g(x), the value of the activation function at x.

Returns — A double containing the value of the derivative of the activation function at x.

G

abstract public double G(double x)
Returns the value of the activation function.

x A double is the point at which the activation function is to be evaluated.

Returns — A double containing the value of the activation function at x.
Description

Standard activation functions are defined as static members of this interface. New activation functions can be defined by implementing a method, \( g(\text{double } x) \), returning the value and a method, \( \text{derivative(double } x, \text{ double } y) \), returning the derivative of \( g \) evaluated at \( x \) where \( y = g(x) \).

See Also

Imsl.DataMining.Neural.Perceptron (p. 706)

Link Class

A link in a neural network.

public class Imsl.DataMining.Neural.Link

Properties

From

virtual public Imsl.DataMining.Neural.Node From {get; }  
The origination Node for this Link. A Node which is the origination Node for this Link.

To

virtual public Imsl.DataMining.Neural.Node To {get; }  
The destination Node for this Link. A Node which is the destination Node for this Link.

Weight

virtual public double Weight {get; set; }  
The weight for this Link. A double which contains the weight attributed to this Link.

Description

Link objects are not created directly. Instead, they are created by factory methods in FeedForwardNetwork.

The method

The method

The method

Each Link object contains an Imsl.DataMining.Neural.Link.Weight (p. 709). Weights are used in computing Imsl.DataMining.Neural.Perceptron (p. 706) values.

**See Also**

Imsl.DataMining.Neural.FeedForwardNetwork (p. 688)

---

**Itrainer Interface**

Interface implemented by classes used to train an Imsl.DataMining.Neural.Network (p. 762).

```csharp
public interface Imsl.DataMining.Neural.ITrainer

Properties

ErrorGradient
abstract public double[] ErrorGradient {get; }
The value of the gradient of the error function with respect to the
Imsl.DataMining.Neural.Link.Weight (p. 709)s. A double array, the length of the
number of Weights, containing the value of the gradient of the error function with respect
to the Weights at the computed optimal point.
Before training, null is returned.

ErrorStatus
abstract public int ErrorStatus {get; }
The error status. An int specifying the error. If there was no error, zero is returned.
A non-zero return indicates a potential problem with the trainer.

ErrorValue
abstract public double ErrorValue {get; }
```
The value of the error function minimized by the trainer. A `double` indicating the final value of the error function from the last training. Before training, `NaN` is returned.

**Method**

**Train**

```java
abstract public void Train(Imsl.DataMining.Neural.Network network, double[, ] xData, double[, ] yData)
```

Trains the neural network using supplied training patterns.

- The number of columns in `xData` must equal the number of nodes in the input layer. Each row of `xData` contains a training pattern.
- The number of columns in `yData` must equal the number of perceptrons in the output layer. Each row of `yData` contains a training pattern.

- **network** A `Network` object, which is the `Network` to be trained.
- **xData** A `double` matrix containing the input training patterns.
- **yData** A `double` matrix containing the output training patterns.

**Description**

The method `Train` is used to adjust the `Imsl.DataMining.Neural.Link.Weight` (p. 709)s in a network to best fit a set of observed data. After a `Network` is trained, the other methods in this interface can be used to check the quality of the fit.

---

**QuasiNewtonTrainer Class**


```java
public class Imsl.DataMining.Neural.QuasiNewtonTrainer implements Field
```

**Field**

- **SUM_OF_SQUARES**

  ```java
  public Imsl.DataMining.Neural.QuasiNewtonTrainer.IError SUM_OF_SQUARES
  ```

  Compute the sum of squares error. The sum of squares error term is \( e(y, \hat{y}) = (y - \hat{y})^2 / 2 \).

  This is the default `IError` object used by `QuasiNewtonTrainer`.
Properties

ErrorGradient

```csharp
virtual public double[] ErrorGradient {get; }
```

The value of the gradient of the error function with respect to the
Imsl.DataMining.Neural.Link.Weight (p. 709)s. A double array whose length is equal to
the number of network weights, containing the value of the gradient of the error function
with respect to the weights.

Before training, null is returned.

ErrorStatus

```csharp
virtual public int ErrorStatus {get; }
```

The error status from the trainer. An int representing the error status from the trainer.

Zero indicates that no errors were encountered during training. Any non-zero value
indicates that some error condition arose during training. In many cases the trainer is
able to recover from these conditions and produce a well-trained network.

### Error Status

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error occurred during training.</td>
</tr>
<tr>
<td>1</td>
<td>The last global step failed to locate a lower point than the current error value. The current solution may be an approximate solution and no more accuracy is possible, or the step tolerance may be too large.</td>
</tr>
<tr>
<td>2</td>
<td>Relative function convergence; both the actual and predicted relative reductions in the error function are less than or equal to the relative function convergence tolerance.</td>
</tr>
<tr>
<td>3</td>
<td>Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or the step tolerance is too big.</td>
</tr>
<tr>
<td>4</td>
<td>Optimizer threw a <code>MinUnconMultiVar.FalseConvergenceException</code>.</td>
</tr>
<tr>
<td>5</td>
<td>Optimizer threw a <code>MinUnconMultiVar.MaxIterationsException</code>.</td>
</tr>
<tr>
<td>6</td>
<td>Optimizer threw a <code>MinUnconMultiVar.UnboundedBelowException</code>.</td>
</tr>
</tbody>
</table>


ErrorValue

```csharp
virtual public double ErrorValue {get; }
```

The final value of the error function. A double representing the final value of the error function from the last training.

Before training, NaN is returned.

FalseConvergenceTolerance

```csharp
virtual public double FalseConvergenceTolerance {get; set; }
```
The false convergence tolerance for the Imsl.DataMining.Neural.ITrainer (p. 710). A double specifying the false convergence tolerance.
Default: 2.22044604925031308e-14.
See Also: Imsl.Math.MinUnconMultiVar.FalseConvergenceTolerance (p. 123)

GradientTolerance
virtual public double GradientTolerance {get; set; }
The gradient tolerance. A double specifying the gradient tolerance.
Default: cube root of machine precision.
See Also: Imsl.Math.MinUnconMultiVar.GradientTolerance (p. 123)

MaximumStepsize
virtual public double MaximumStepsize {get; set; }
The maximum step size. A nonnegative double value specifying the maximum allowable step size in the optimizer.
The value of MaximumStepsize will be equal to -999.0 if the default value is to be used and the Imsl.DataMining.Neural.QuasiNewtonTrainer.Train(Imsl.DataMining.Neural.Network,System.Double[0:,0:],System.Double[0:,0:]) (p. 714) method has not been called.
See Also: (p. 124)

MaximumTrainingIterations
virtual public int MaximumTrainingIterations {get; set; }
The maximum number of iterations to use in a training. An int representing the maximum number of training iterations.
Default: 100.
See Also: (p. 124)

RelativeTolerance
virtual public double RelativeTolerance {get; set; }
The relative tolerance. A double representing the relative error tolerance.
It must be in the interval [0,1]. Its default value is 3.66685e-11.
See Also: Imsl.Math.MinUnconMultiVar.RelativeTolerance (p. 124)

StepTolerance
virtual public double StepTolerance {get; set; }
The scaled step tolerance. A double which is the step tolerance.
The second stopping criterion for Imsl.Math.MinUnconMultiVar (p. 122), the optimizer used by this Imsl.DataMining.Neural.ITrainer (p. 710), is that the scaled distance between the last two steps be less than the step tolerance.
Default: 3.66685e-11.
See Also: Imsl.Math.MinUnconMultiVar.StepTolerance (p. 124)
TrainingIterations

    virtual public int TrainingIterations {get; }
    The number of iterations used during training. An int representing the number of
    iterations used during training.

See Also: (p. 124)

Constructor

QuasiNewtonTrainer

    public QuasiNewtonTrainer()
    Constructs a QuasiNewtonTrainer object.

Methods

GetError

    virtual public Imsl.DataMining.Neural.QuasiNewtonTrainer.IError GetError()
    Returns the function used to compute the error to be minimized.
    Returns — The IError object containing the function to be minimized.

SetError

    virtual public void SetError(Imsl.DataMining.Neural.QuasiNewtonTrainer.IError error)
    Sets the function that computes the network error.
    The default is to compute the sum of squares error, SUM_OF_SQUARES.

    error The IError object containing the function to be used to compute the network
    error.

Train

    virtual public void Train(Imsl.DataMining.Neural.Network network, double[,] xData, double[,] yData)
    Trains the neural network using supplied training patterns.
    The number of columns in xData must equal the number of Imsl.DataMining.Neural.Node
    (p. 704)’s in the input layer.
    The number of columns in yData must equal the number of
    Imsl.DataMining.Neural.Perceptron (p. 706)’s in the output layer.
    Each row of xData and yData contains a training pattern. The number of rows in these
    two arrays must be at least equal to the number of Imsl.DataMining.Neural.Link.Weight
    (p. 709)’s in the Network.

    network The Network to be trained.
    xData An input double matrix containing training patterns.
    yData An output double matrix containing output training patterns.
See Also

Imsl.Math.MinUnconMultiVar (p. 122)

QuasiNewtonTrainer.IError Interface

Error function to be minimized by trainer.

public interface Imsl.DataMining.Neural.QuasiNewtonTrainer.IError

Methods

Error

abstract public double Error(double computed, double expected)
The contribution to the error from a single training output target. This is the function 
\( e(y_i, \hat{y}_i) \).

computed A double representing the computed value.

expected A double representing the expected value.

Returns — A double representing the contribution to the error from a single training output target.

ErrorGradient

abstract public double ErrorGradient(double computed, double expected)
The derivative of the error function with respect to the forecast output.

computed A double representing the computed value.

expected A double representing the expected value.

Returns — A double representing the derivative of the error function with respect to the forecast output.

Description

This trainer attempts to solve the problem

\[
\min_w \sum_{i=0}^{n-1} e(y_i, \hat{y}_i)
\]

where \( w \) are the weights, \( n \) is the number of training patterns, \( y_i \) is a training target output and \( \hat{y}_i \) is its forecast value.

This interface defines the function \( e(y, \hat{y}) \) and its derivative with respect to its computed value, \( de/d\hat{y} \).
LeastSquaresTrainer Class

Trains a Imsl.DataMining.Neural.FeedForwardNetwork (p. 688) using a Levenberg-Marquardt algorithm for minimizing a sum of squares error.

public class Imsl.DataMining.Neural.LeastSquaresTrainer implements

Properties

ErrorGradient

virtual public double[] ErrorGradient {get; }
The value of the gradient of the error function with respect to the Imsl.DataMining.Neural.Link.Weight (p. 709)s. A double array whose length is equal to the number of network Weights, containing the value of the gradient of the error function with respect to the Weights.
Before training, null is returned.

ErrorStatus

virtual public int ErrorStatus {get; }
The error status from the trainer. An int which contains the error status.
Zero indicates that no errors were encountered during training. Any non-zero value indicates that some error condition arose during training.
In many cases the trainer is able to recover from these conditions and produce a well-trained network.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All convergence tests were met.</td>
</tr>
<tr>
<td>1</td>
<td>Scaled step tolerance was satisfied. The current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or StepTolerance is too big.</td>
</tr>
<tr>
<td>2</td>
<td>Scaled actual and predicted reductions in the function are less than or equal to the relative function convergence tolerance RelativeTolerance.</td>
</tr>
<tr>
<td>3</td>
<td>Iterates appear to be converging to a noncritical point. Incorrect gradient information, a discontinuous function, or stopping tolerances being too tight may be the cause.</td>
</tr>
<tr>
<td>4</td>
<td>Five consecutive steps with the maximum stepsize have been taken. Either the function is unbounded below, or has a finite asymptote in some direction, or the maximum stepsize is too small.</td>
</tr>
<tr>
<td>5</td>
<td>Too many iterations required.</td>
</tr>
</tbody>
</table>
ErrorValue

    virtual public double ErrorValue {get; }

    The final value of the error function. A double containing the final value of the error
    function from the last training.
    
    Before training, NaN is returned.

FalseConvergenceTolerance

    virtual public double FalseConvergenceTolerance {get; set; }

    The false convergence tolerance. A double specifying the false convergence tolerance.
    
    Default: 1.0e-14.
    
    See Also: NonlinLeastSquares.FalseConvergenceTolerance (p. 130)

GradientTolerance

    virtual public double GradientTolerance {get; set; }

    The gradient tolerance. A double specifying the gradient tolerance.
    
    Default: 2.0e-5.
    
    See Also: NonlinLeastSquares.GradientTolerance (p. 130)

InitialTrustRegion

    virtual public double InitialTrustRegion {get; set; }

    The initial trust region. A double which specifies the initial trust region radius.
    
    Default: unlimited trust region.
    
    The value of InitialTrustRegion will be equal to -999.0 if the default value is to be
    used and the Train method has not been called.
    
    See Also: NonlinLeastSquares.InitialTrustRegion (p. 130)

MaximumStepsize

    virtual public double MaximumStepsize {get; set; }

    The maximum step size. A nonnegative double value specifying the maximum allowable
    stepsize in the optimizer.
    
    Default: $10^{3}||w||_2$, where $w$ are the values of the Imsl.DataMining.Neural.Link.Weight (p. 709)s in the network when training starts.
    
    The value of MaximumStepsize will be equal to -999.0 if the default value is to be used
    and the Train method has not been called.
    
    See Also: NonlinLeastSquares.MaximumStepsize (p. 130)

MaximumTrainingIterations

    virtual public int MaximumTrainingIterations {get; set; }

    The maximum number of iterations used by the nonlinear least squares solver. An int
    which specifies the maximum number of iterations to be used by the nonlinear least
    squares solver.
    
    Its default value is 1000.
    
    See Also: NonlinLeastSquares.RelativeTolerance (p. 131)
RelativeTolerance

virtual public double RelativeTolerance {get; set; }

The relative tolerance. A double which specifies the relative error tolerance.

It must be in the interval [0,1]. Its default value is 1.0e-20.

See Also: NonlinLeastSquares.RelativeTolerance (p. 131)

StepTolerance

virtual public double StepTolerance {get; set; }

The step tolerance used to step between Imsl.DataMining.Neural.Link.Weight (p. 709)s.

A double which specifies the scaled step tolerance to use when changing the weights.

Default: 1.0e-5.

See Also: NonlinLeastSquares.StepTolerance (p. 131)

Constructor

LeastSquaresTrainer

public LeastSquaresTrainer()

Creates a LeastSquaresTrainer.

Method

Train

virtual public void Train(Imsl.DataMining.Neural.Network network, double[,] xData, double[,] yData)

Trains the neural network using supplied training patterns.

Each row of xData and yData contains a training pattern. These number of rows in two arrays must be equal.

network The Network to be trained.

xData A double matrix which contains the input training patterns. The number of columns in xData must equal the number of Imsl.DataMining.Neural.Node (p. 704)s in the Imsl.DataMining.Neural.InputLayer (p. 700).

yData A double matrix which contains the output training patterns. The number of columns in yData must equal the number of Imsl.DataMining.Neural.Perceptron (p. 706)s in the Imsl.DataMining.Neural.OutputLayer (p. 702).

See Also

NonlinLeastSquares (p. 129)
EpochTrainer Class

Two-stage training using randomly selected training patterns in stage I.

```java
public class Imsl.DataMining.Neural.EpochTrainer implements Properties

Properties

EpochSize
   virtual public int EpochSize {get; set; }
   The number of randomly selected training patterns in each stage I epoch. An int which
   contains the number of sample training patterns in each stage I epoch.

ErrorGradient
   virtual public double[] ErrorGradient {get; }
   The value of the gradient of the error function with respect to the
   Imsl.DataMining.Neural.Network.Weights (p. 763). A double array whose length is
   equal to the number of Network.Weights, containing the value of the gradient of the
   error function with respect to the weights.
   Before training, null is returned.

ErrorStatus
   virtual public int ErrorStatus {get; }
   The training error status. An int containing the error status from stage II.
   If there is no stage II then the number of stage I epochs that returned a non-zero error
   status is returned.

ErrorValue
   virtual public double ErrorValue {get; }
   The value of the error function. A double containing final value of the error function
   from the last training. Before training, NaN is returned.

NumberOfEpochs
   virtual public int NumberOfEpochs {get; set; }
   The number of epochs used during stage I training. An int which contains the number
   of epochs used during stage I training.

Random
   virtual public Imsl.Stat.Random Random {get; set; }
   The random number generator used to perturb the stage I guesses. The Random object
   used to generate stage I perturbations.
```
Constructors

EpochTrainer
public EpochTrainer(Imsl.DataMining.Neural.ITrainer stage1Trainer)
Creates a single stage EpochTrainer. Stage II training is bypassed.

stage1Trainer The ITrainer used in stage I.

EpochTrainer
public EpochTrainer(Imsl.DataMining.Neural.ITrainer stage1Trainer, Imsl.DataMining.Neural.ITrainer stage2Trainer)
Creates a two-stage EpochTrainer.

stage1Trainer The stage I ITrainer.
stage2Trainer The stage II ITrainer, or null if stage II is to be bypassed.

Methods

SetRandomSamples
Sets the random number generators used to select random training patterns in stage I.
The two random number generators should be independent.

randomA A Random object which is the first random number generator.
randomB A Random object which is the second random number generator, independent of randomA.

Train
virtual public void Train(Imsl.DataMining.Neural.Network network, double[,] xData, double[,] yData)
Trains the neural network using supplied training patterns.

Each row of xData and yData contains a training pattern. These number of rows in two
arrays must be equal.

network The Network to be trained.
xData A double matrix specifying the input training patterns. The number of columns in xData must equal the number of Imsl.DataMining.Neural.Node (p. 704)s in the Imsl.DataMining.Neural.InputLayer (p. 700).

yData A double containing the output training patterns. The number of columns in yData must equal the number of Imsl.DataMining.Neural.Perceptron (p. 706)s in the Imsl.DataMining.Neural.OutputLayer (p. 702).
Description

The **EpochTrainer**, is a meta-trainer that combines two trainers. The first trainer is used on a series of randomly selected subsets of the training patterns. For each subset, the Imsl.DataMining.Neural.Network.Weights (p. 763) are initialized to their initial values plus a random offset.

Stage II then refines the result found in stage I. The best result from the stage I trainings is used as the initial guess with the second trainer operating on the full set of training patterns. Stage II is optional, if the second trainer is null then the best stage I result is returned as the **EpochTrainer**’s result.

---

**ScaleFilter Class**

Scales or unscales continuous data prior to its use in neural network training, testing, or forecasting.

```
public class Imsl.DataMining.Neural.ScaleFilter
```

**Properties**

**Center**

```
    virtual public double Center {get; set; }
```

The measure of center to be used during z-score scaling. A double containing the measure of center to be used during scaling. If this property is not set then the measure of center is computed from the data.

**Spread**

```
    virtual public double Spread {get; set; }
```

The measure of spread to be used during z-score scaling. A double containing the measure of spread to be used during z-score scaling. If this property is not set then the measure of spread is computed from the data.

**Constructor**

```
    public ScaleFilter(Imsl.DataMining.Neural.ScaleFilter.ScalingMethod
                        scalingMethod)
```

scalingMethod An int specifying the scaling method to be applied.

Methods

Decode

virtual public void Decode(int columnIndex, double[,] z)
Unscales a single column of a two dimensional array of values.
Indexing is zero-based.
Its columnIndex-th column is modified in place.

columnIndex An int specifying the index of the column of z to unscale.
z A double matrix containing the values to be unscaled.

Decode

virtual public double[] Decode(double[] z)
Unscales an array of values.

z A double array of values to be unscaled.

Returns — A double array containing the filtered data.

Decode

virtual public double Decode(double z)
Unscales a value.

z A double containing the value to be unscaled.

Returns — A double containing the filtered data.

Encode

virtual public void Encode(int columnIndex, double[,] x)
Scales a single column of a two dimensional array of values.
Indexing is zero-based.
Its columnIndex-th column is modified in place.

columnIndex An int specifying the index of the column of x to scale.
x A double matrix containing the value to be scaled.

Encode

virtual public double[] Encode(double[] x)
Scales an array of values.
x  A double array containing the data to be scaled.

Returns — A double array containing the scaled data.

Encode
    virtual public double Encode(double x)
    Scales a value.

    x  A double containing the value to be scaled.

    Returns — A double containing the scaled value.

GetBounds
    virtual public double[] GetBounds()
    Retrieves bounds used during bounded scaling.

    i  result[b]
    0  realMin. Lowest expected value in the data to be filtered.
    1  realMax. Largest expected value in the data to be filtered.
    2  targetMin. Lowest allowed value in the filtered data.
    3  targetMax. Largest allowed value in the filtered data.

Returns — A double array of length 4 containing the bounds.

SetBounds
    virtual public void SetBounds(double realMin, double realMax, double targetMin, double targetMax)
    Sets bounds to be used during bounded scaling and unscaling.

    This method is normally called prior to calls to
    Imsl.DataMining.Neural.ScaleFilter.Encode(System.Double) (p. 723) or
    Imsl.DataMining.Neural.ScaleFilter.Decode(System.Double) (p. 722). Otherwise the
default bounds are realMin = 0, realMax = 1, targetMin = 0, and targetMax = 1. These
bounds are ignored for unbounded scaling.

    realMin  A double containing the lowest expected value in the data to be filtered.
    realMax  A double containing the largest expected value in the data to be filtered.
    targetMin  A double containing the lowest allowed value in the filtered data.
    targetMax  A double containing the largest allowed value in the filtered data.

Description

Bounded scaling is used to ensure that the values in the scaled array fall between a lower and
upper bound. The scale limits have the following interpretation:
### Argument Interpretation

<table>
<thead>
<tr>
<th>Argument</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>realMin</td>
<td>The lowest value expected in x.</td>
</tr>
<tr>
<td>realMax</td>
<td>The largest value expected in x.</td>
</tr>
<tr>
<td>targetMin</td>
<td>The lower bound for the values in the scaled data.</td>
</tr>
<tr>
<td>targetMax</td>
<td>The upper bound for the values in the scaled data.</td>
</tr>
</tbody>
</table>

The scale limits are set using the method

\[
\]

(p. 723).

The specific scaling used is controlled by the argument `scalingMethod` used when constructing the filter object. If `scalingMethod` is `ScalingMethod.None`, then no scaling is performed on the data.

If the input parameter `scalingMethod` is `ScaleMethod.Bounded` then the bounded method of scaling and unscaling is applied to \( x \). The scaling operation is conducted using the scale limits set in method `SetBounds`, using the following calculation:

\[
z = r(x - \text{realMin}) + \text{targetMin},
\]

where

\[
r = \frac{\text{targetMax} - \text{targetMin}}{\text{realMax} - \text{realMin}}.
\]

If `scalingMethod` is one of `UnboundedZScoreMeanStdev`, `UnboundedZScoreMedianMAD`, `BoundedZScoreMeanStdev`, or `BoundedZScoreMedianMAD`, then the z-score method of scaling is used. These calculations are based upon the following scaling calculation:

\[
z = \frac{(x - a)}{b},
\]

where \( a \) is a measure of center for \( x \), and \( b \) is a measure of the spread of \( x \).

If `scalingMethod` is `UnboundedZScoreMeanStdev` or `BoundedZScoreMeanStdev`, then \( a \) and \( b \) are the arithmetic average and sample standard deviation of the training data.

If `scalingMethod` is `UnboundedZScoreMedianMAD` or `BoundedZScoreMedianMAD`, then \( a \) and \( b \) are the median and \( \hat{s} \), where \( \hat{s} \) is a robust estimate of the population standard deviation:

\[
\hat{s} = \frac{\text{MAD}}{0.6745}
\]

where MAD is the Mean Absolute Deviation

\[
\text{MAD} = \text{median}\{|x - \text{median}(x)|\}
\]

The Mean Absolute Deviation is a robust measure of spread calculated by finding the median of the absolute value of differences between each non-missing value for the \( i \)-th variable and the median of those values.
If the method Imsl.DataMining.Neural.ScaleFilter.Decode(System.Double) (p. 722) is called then an unscaling operation is conducted by inverting using:

\[ x = \frac{(z - \text{targetMin})}{r} + \text{realMin}. \]

**Unbounded z-score Scaling**

If \textit{scalingMethod} is \texttt{UnboundedZScoreMeanStdev} or \texttt{UnboundedZScoreMedianMAD}, then a scaling operation is conducted using the z-score calculation:

\[ z = \frac{(x - \text{center})}{\text{spread}}. \]

If \textit{scalingMethod} is \texttt{UnboundedZScoreMeanStdev} then
Imsl.DataMining.Neural.ScaleFilter.Center (p. 721) is set equal to the arithmetic average \( \bar{x} \) of \( x \), and Imsl.DataMining.Neural.ScaleFilter.Spread (p. 721) is set equal to the sample standard deviation of \( x \). If \textit{scalingMethod} is \texttt{UnboundedZScoreMedianMAD} then \texttt{Center} is set equal to the median \( \tilde{m} \) of \( x \), and \texttt{Spread} is set equal to the Mean Absolute Difference (MAD).

The method \texttt{Decode} can be used to unfilter data using the inverse calculation for the above equation:

\[ x = \text{spread} \cdot z + \text{center}. \]

**Bounded z-score Scaling**

This method is essentially the same as the z-score calculation described above with additional scaling or unscaling using the scale limits set in method \texttt{SetBounds}. The scaling operation is conducted using the well known z-score calculation:

\[ z = \frac{r \cdot (x - \text{center})}{\text{spread}} - r \cdot \text{realMin} + \text{targetMin}. \]

If \textit{scalingMethod} is \texttt{UnboundedZScoreMeanStdev} then \texttt{Center} is set equal to the arithmetic average \( \bar{x} \) of \( x \), and \texttt{Spread} is set equal to the sample standard deviation of \( x \). If \textit{scalingMethod} is \texttt{UnboundedZScoreMedianMAD} then \texttt{Center} is set equal to the median \( \tilde{m} \) of \( x \), and \texttt{Spread} is set equal to the Mean Absolute Difference (MAD).

The method \texttt{Decode} can be used to unfilter data using the inverse calculation for the above equation:

\[ x = \frac{\text{spread} \cdot (z - \text{targetMin})}{r} + \text{spread} \cdot \text{realMin} + \text{center} \]
Example: ScaleFilter

In this example three sets of data, $X_0$, $X_1$, and $X_2$ are scaled using the methods described in the following table:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_0$</td>
<td>0</td>
<td>No Scaling</td>
</tr>
<tr>
<td>$X_1$</td>
<td>4</td>
<td>Bounded Z-score scaling using the mean and standard deviation of $X_1$</td>
</tr>
<tr>
<td>$X_2$</td>
<td>5</td>
<td>Bounded Z-score scaling using the median and MAD of $X_2$</td>
</tr>
</tbody>
</table>

The bounds, measures of center and spread for $X_1$ and $X_2$ are:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Real Limits</th>
<th>Target Limits</th>
<th>Measure of Center</th>
<th>Measure of Spread</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>(-6, +6)</td>
<td>(-3, +3)</td>
<td>3.4 (Mean)</td>
<td>1.7421 (Std. Dev.)</td>
</tr>
<tr>
<td>$X_2$</td>
<td>(-3, +3)</td>
<td>(-3, +3)</td>
<td>2.4 (Median)</td>
<td>1.3343 (MAD/0.6745)</td>
</tr>
</tbody>
</table>

The real and target limits are used for bounded scaling. The measures of center and spread are used to calculate z-scores. Using these values for $x_1[0]=3.5$ yields the following calculations:

For $x_1[0]$, the scale factor is calculated using the real and target limits in the above table:

$$r = \frac{3-(-3)}{6-(-6)} = 0.5$$

The z-score for $x_1[0]$ is calculated using the measures of center and spread:

$$z_1[0] = \frac{3.5 - 3.4}{1.7421} = 0.057402$$

Since method=4 is used for $x_1$, this z-score is bounded (scaled) using the real and target limits:

$$z_1(\text{bounded}) = r(z_1[0]) - r(\text{realMin}) + (\text{targetMin}) = 0.5(0.057402) - 0.5(-6) + (-3) = 0.029$$

The calculations for $x_2[0]$ are nearly identical, except that since method=5 for $x_2$, the median and MAD replace the mean and standard deviation used to calculate $z_1(\text{bounded})$:

$$r = \frac{3-(-3)}{3-(-3)} = 1,$$

$$z_2[0] = \frac{3.1 - 2.4}{1.3343} = 0.525,$$

$$z_2(\text{bounded}) = r(z_2[0]) - r(\text{realMin}) + (\text{targetMin}) = 1(0.525) - 1(-3) + (-3) = 0.525$$

using System;
using Imsl.Stat;
using Imsl.Math;
using Imsl.DataMining.Neural;

public class ScaleFilterEx1
{

public static void Main(System.String[] args)
{
    ScaleFilter[] scaleFilter = new ScaleFilter[3];
    scaleFilter[0] = new ScaleFilter(ScaleFilter.ScalingMethod.None);
    scaleFilter[1] = new ScaleFilter(ScaleFilter.ScalingMethod.BoundedZScoreMeanStdev);
    scaleFilter[1].SetBounds(- 6.0, 6.0, - 3.0, 3.0);
    scaleFilter[2] = new ScaleFilter(ScaleFilter.ScalingMethod.BoundedZScoreMedianMAD);
    scaleFilter[2].SetBounds(- 3.0, 3.0, - 3.0, 3.0);
    double[] y0, y1, y2;
    double[] x0 = new double[]{1.2, 0.0, - 1.4, 1.5, 3.2};
    double[] x1 = new double[]{3.5, 2.4, 4.4, 5.6, 1.1};
    double[] x2 = new double[]{3.1, 1.5, - 1.5, 2.4, 4.2};
    // Perform forward filtering
    y0 = scaleFilter[0].Encode(x0);
    y1 = scaleFilter[1].Encode(x1);
    y2 = scaleFilter[2].Encode(x2);
    // Display x0
    System.Console.Out.Write("X0 = { ");
    for (int i = 0; i < 4; i++)
        System.Console.Out.Write(x0[i] + ", ");
    // Display summary statistics for X1
    System.Console.Out.WriteLine("X1 Mean: " + scaleFilter[1].Center);
    System.Console.Out.WriteLine("X1 Std. Dev.: " + scaleFilter[1].Spread);
    // Display summary statistics for X2
    System.Console.Out.WriteLine("X2 Median: " + scaleFilter[2].Center);
    System.Console.Out.WriteLine("X2 MAD/0.6745: " + scaleFilter[2].Spread);
    // Perform inverse filtering
    double[] z0, z1, z2;
    z0 = scaleFilter[0].Decode(y0);
    z1 = scaleFilter[1].Decode(y1);
    z2 = scaleFilter[2].Decode(y2);
    System.Console.Out.WriteLine("Decoded Z0");
    System.Console.Out.WriteLine("Decoded Z1");
    System.Console.Out.WriteLine("Decoded Z2");
}
pm.Print(z2);
}
}

Output

X0 = {1.2, 0, -1.4, 1.5, 3.2}
X1 = {3.5, 2.4, 4.4, 5.6, 1.1}
X1 Mean: 3.4
X1 Std. Dev.: 1.74212513901843
X2 = {3.1, 1.5, -1.5, 2.4, 4.2}
X2 Median: 2.4
X2 MAD/0.6745: 1.3343199665504

Filtered X0 Using Method=0 (no scaling)
  0
  1 1.2
  2 -1.4
  3 1.5
  4 3.2

Filtered X1 Using Bounded Z-score Scaling
  with Center=Mean and Spread=Std. Dev.
  0
  1 -0.287005788965146
  2 0.287005788965146
  3 0.631412735723321
  4 -0.660113314619835

Filtered X2 Using Bounded Z-score Scaling
  with Center=Median and Spread=MAD/0.6745
  0
  1 -0.67449750196082
  2 -2.9278891751635
  3 0
  4 1.34897950039216

Decoded Z0
  0
  1 1.2
  2 0
  3 -1.4
  4 3.2

Decoded Z1
  0
  1 3.5
ScaleFilter.ScalingMethod Enumeration

Scaling Method

public enumeration Imsl.DataMining.Neural.ScaleFilter.ScalingMethod

Fields

Bounded
   public Imsl.DataMining.Neural.ScaleFilter.ScalingMethod Bounded
   Flag to indicate bounded scaling.

BoundedZScoreMeanStdev
   public Imsl.DataMining.Neural.ScaleFilter.ScalingMethod
   BoundedZScoreMeanStdev
   Flag to indicate bounded z-score scaling using the mean and standard deviation.

BoundedZScoreMedianMAD
   public Imsl.DataMining.Neural.ScaleFilter.ScalingMethod
   BoundedZScoreMedianMAD
   Flag to indicate bounded z-score scaling using the median and mean absolute difference.

None
   public Imsl.DataMining.Neural.ScaleFilter.ScalingMethod None
   Flag to indicate no scaling.

UnboundedZScoreMeanStdev
   public Imsl.DataMining.Neural.ScaleFilter.ScalingMethod
   UnboundedZScoreMeanStdev
   Flag to indicate unbounded z-score scaling using the mean and standard deviation.

UnboundedZScoreMedianMAD
UnsupervisedNominalFilter Class

Converts nominal data into a series of binary encoded columns for input to a neural network. It also reverses the aforementioned encoding, accepting binary encoded data and returns an array of integers representing the classes for a nominal variable.

```csharp
public class Imsl.DataMining.Neural.UnsupervisedNominalFilter
```

**Property**

**NumberOfClasses**

```csharp
virtual public int NumberOfClasses { get; }
```

The number of classes in the nominal variable. An `int` containing the number of classes in the nominal variable.

**Constructor**

```csharp
public UnsupervisedNominalFilter(int nClasses)
```

Constructor for `UnsupervisedNominalFilter`.

**nClasses** An `int` specifying the number of categories in the nominal variable to be filtered.

**Methods**

**Decode**

```csharp
virtual public int[] Decode(int[,] z)
```

Decodes a matrix representing the binary encoded columns of the nominal variable.

This is the inverse of the `Imsl.DataMining.Neural.UnsupervisedNominalFilter.Encode(System.Int32[])` (p. 731) method.

**z** An `int` matrix containing the data to be decoded.

Returns — An `int` array containing the decoded data.
**Decode**

\[
\text{virtual public int Decode(int[]} z) \\
\text{Decodes a binary encoded array into its nominal category.} \\
\text{This is the inverse of the} \\
\text{Imsl.DataMining.Neural.UnsupervisedNominalFilter.Encode(System.Int32][}) \text{ (p. 731)} \\
\text{method.} \\
z \text{ An int array containing the data to be decoded.} \\
\text{Returns} — \text{An int containing the number associated with the category encoded in } z.
\]

**Encode**

\[
\text{virtual public int[] Encode(int x)} \\
\text{Apply forward encoding to a value.} \\
\text{Class number must be in the range 1 to } n\text{Classes.} \\
x \text{ An int containing the value to be encoding.} \\
\text{Returns} — \text{An int array containing the encoded data.}
\]

\[
\text{virtual public int[,] Encode(int[] x)} \\
\text{Encodes class data prior to its use in neural network training.} \\
\text{Class number must be in the range 1 to } n\text{Classes.} \\
x \text{ An int array containing the data to be encoded.} \\
\text{Returns} — \text{An int matrix containing the encoded data.}
\]

**Description**

**Binary Encoding**

Method Imsl.DataMining.Neural.UnsupervisedNominalFilter.Encode(System.Int32[]) (p. 731) can be used to apply binary encoding. Referring to the result as \( z \), binary encoding takes each category in the nominal variable \( x \), and creates a column in \( z \) containing all zeros and ones. A value of zero indicates that this category was not present and a value of one indicates that it is present.

For example, if \( x[] = \{2,1,3,4,2,4\} \) then \( n\text{Classes} = 4 \), and

\[
\begin{array}{cccc}
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{array}
\]

Notice that the number of columns in the result, \( z \), is equal to the number of distinct classes in \( x \). The number of rows in \( z \) is equal to the length of \( x \).
Binary Decoding

Unfiltering can be performed using the method

\texttt{Imsl.DataMining.Neural.UnsupervisedNominalFilter.Decode(System.Int32[])} \texttt{(p. 730)}. In this case, \( z \) is the input, and we refer to \( x \) as the output. Binary unfiltering takes binary representation in \( z \), and returns the appropriate class in \( x \).

For example, if a row in \( z \) equals \( \{0, 1, 0, 0\} \), then the return value from \texttt{Decode} would be 2 for that row. If a row in \( z \) equals \( \{1, 0, 0, 0\} \), then the return value from \texttt{Decode} would be 1 for that row. Notice these are the same values as the first two elements of the original \( x \) because classes are numbered sequentially from 1 to \( nClasses \). This ensures that the results of \texttt{Decode} are associated with the \( i \)-th class in \( x \).

Example: \texttt{UnsupervisedNominalFilter}

In this example a data set with 7 observations and 3 classes is filtered.

```csharp
using System;
using Imsl.Stat;
using Imsl.Math;
using Imsl.DataMining.Neural;

class UnsupervisedNominalFilterEx1
{
    [STAThread]
    public static void Main(string[] args)
    {
        int nClasses = 3;
        UnsupervisedNominalFilter filter = new UnsupervisedNominalFilter(nClasses);
        int nObs = 7;
        int[] x = new int[]{3, 3, 1, 2, 2, 1, 2};
        int[] xBack = new int[nObs];
        int[,] z;

        // Perform Binary Filtering.
        z = filter.Encode(x);
        PrintMatrix pm = new PrintMatrix();
        pm.SetTitle("Filtered x");
        pm.Print(z);

        // Perform Binary Un-filtering.
        int[] tmp = new int[z.GetLength(1)];
        for (int i = 0; i < nObs; i++)
        {
            for (int j = 0; j < z.GetLength(1); j++)
                tmp[j] = z[i, j];
            xBack[i] = filter.Decode(tmp);
        }
        pm.SetTitle("Result of inverse filtering");
        pm.Print(xBack);
    }
}
```
Output

Filtered x
0 1 2
0 0 0 1
1 0 0 1
2 1 0 0
3 0 1 0
4 0 1 0
5 1 0 0
6 0 1 0

Result of inverse filtering
0
0 3
1 3
2 1
3 2
4 2
5 1
6 2

UnsupervisedOrdinalFilter Class

Encodes ordinal data into percentages for input to a neural network. It also allows decoding, accepting a percentage and converting it into an ordinal value.

```csharp
public class Imsl.DataMining.Neural.UnsupervisedOrdinalFilter
```

Properties

NumberOfClasses

```csharp
virtual public int NumberOfClasses {get; }
```

The number of categories associated with this ordinal variable. An `int` containing the number of categories associated with this ordinal variable.

Percentages

```csharp
virtual public double[] Percentages {get; set; }
```

The cumulative percentages used during encoding and decoding. A `double` array of length `nClasses` containing the cumulative percentages associated with the ordinal categories.
If a transform has been applied to the percentages then the transformed percentages are returned. Setting untransformed cumulative percentages with this method bypasses calculating cumulative percentages based on the data being encoded. The percentages must be nondecreasing in the interval [0, 100], with the last element equal to 100. If this method is used it must be called prior to any calls to the encoding and decoding methods.

**Transform**

```csharp
virtual public int Transform {get; }
```

The transform flag used for encoding and decoding. An `int` containing the transform flag used for encoding and decoding.

**Constructor**

```csharp
public UnsupervisedOrdinalFilter(int nClasses,
```

Constructor for `UnsupervisedOrdinalFilter`.

Values for `Imsl.DataMining.Neural.UnsupervisedOrdinalFilter.Transform` (p. 734) are:

- `nClasses` An `int` specifying the number of classes in the data to be filtered.
- `transform` An `TransformMethod` specifying the transform to be applied to the percentages.

**Methods**

**Decode**

```csharp
virtual public int[] Decode(double[] y)
```

Decodes an array of encoded ordinal values.

- `y` A `double` array containing the encoded ordinal data to be decoded.

Returns — An `int` array containing the decoded ordinal classifications.

**Decode**

```csharp
virtual public int Decode(double y)
```

Decodes an encoded ordinal variable.

- `y` A `double` containing the encoded value to be decoded.

Returns — An `int` containing the ordinal category associated with `y`.
Encode

virtual public double Encode(int x)
Encodes an ordinal category.

x must be an integer between 1 and nClasses.

x  An int containing the ordinal category.

Returns — A double containing the encoded value, a transformed cumulative percentage.

Encode

virtual public double[] Encode(int[] x)
Encodes an array of ordinal categories into an array of transformed percentages.

Categories must be numbered from 1 to nClasses.

x  An int array containing the categories for the ordinal variable.

Returns — A double array of the transformed percentages.

Description

Class UnsupervisedOrdinalFilter is designed to either encode or decode ordinal variables. Encoding consists of transforming the ordinal classes into percentages, with each percentage being equal to the percentage of the data at or below this class.

Ordinal Encoding

In this case, x is input to the method
Imsl.DataMining.Neural.UnsupervisedOrdinalFilter.Encode(System.Int32[]) (p. 735) and is filtered by converting each ordinal class value into a cumulative percentage.

For example, if x[]=\{2,1,3,4,2,4,1,1,3,3\} then nClasses =4, and Encode returns the ordinal class designation with the cumulative percentages displayed in the following table. Cumulative percentages are equal to the percent of the data in this class or a lower class.

<table>
<thead>
<tr>
<th>Ordinal Class</th>
<th>Frequency</th>
<th>Cumulative Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>30%</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>80%</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>100%</td>
</tr>
</tbody>
</table>

Classes in x must be numbered from 1 to nClasses.

The values returned from encoding or decoding depend upon the setting of
Imsl.DataMining.Neural.UnsupervisedOrdinalFilter.Transform (p. 734). In this example, if the
filter was constructed with \texttt{Transform = TransformMethod.None}, then the method \texttt{Encode} will return

\[ z[] = \{50, 30, 80, 100, 50, 100, 30, 30, 80, 80\} .\]

If the filter was constructed with \texttt{Transform = TransformMethod.Sqrt}, then the square root of these values is returned; i.e.,

\[ z[i] = \sqrt{\frac{z[i]}{100}} \]

\[ z[] = \{0.71, 0.55, 0.89, 1.0, 0.71, 1.0, 0.55, 0.55, 0.89, 0.89\} ;\]

If the filter was constructed with \texttt{Transform = TransformMethod.AsinSqrt}, then the arcsin square root of these values is returned using the following calculation:

\[ z[i] = \arcsin \left( \sqrt{\frac{z[i]}{100}} \right) \]

**Ordinal Decoding**

Ordinal decoding takes a transformed cumulative proportion and converts it into an ordinal class value.

**Example: UnsupervisedOrdinalFilter**

In this example a data set with 10 observations and 4 classes is filtered.

```csharp
using System;
using Imsl.Stat;
using Imsl.Math;
using Imsl.DataMining.Neural;

public class UnsupervisedOrdinalFilterEx1
{
    [STAThread]
    public static void Main(System.String[] args)
    {
        int nClasses = 4;
        UnsupervisedOrdinalFilter filter = new UnsupervisedOrdinalFilter(nClasses, UnsupervisedOrdinalFilter.TransformMethod.AsinSqrt);
        int[] x = new int[]{2, 1, 3, 4, 2, 4, 1, 1, 3, 3};
        int nObs = x.Length;
        int[] xBack;
        double[] z;
```
// Ordinal Filtering.
z = filter.Encode(x);
// Print result without row/column labels.
PrintMatrix pm = new PrintMatrix();
PrintMatrixFormat mf;
mf = new PrintMatrixFormat();
mf.SetNoRowLabels();
mf.SetNoColumnLabels();
pm.SetTitle("Filtered data");
pm.Print(mf, z);

// Ordinal Un-filtering.
pm.SetTitle("Un-filtered data");
xBack = filter.Decode(z);

// Print results of Un-filtering.
pm.Print(mf, xBack);
}

Output

Filtered data

0.785398163397448
0.579639740363704
1.10714871779409
1.5707963267949
0.785398163397448
1.5707963267949
0.579639740363704
0.579639740363704
1.10714871779409
1.10714871779409

Un-filtered data

2
1
3
4
2
4
1
1
3
3
UnsupervisedOrdinalFilter.TransformMethod

Enumeration

Transform type

public enumeration
Imsl.DataMining.Neural.UnsupervisedOrdinalFilter.TransformMethod

Fields

AsinSqrt

AsinSqrt
Flag to indicate the arcsine square root transform will be applied to the percentages.

None

None
Flag to indicate no transformation of percentages.

Sqrt

Sqrt
Flag to indicate the square root transform will be applied to the percentages.

TimeSeriesFilter Class

Converts time series data to a lagged format used as input to a neural network.

public class Imsl.DataMining.Neural.TimeSeriesFilter

Constructor

TimeSeriesFilter

public TimeSeriesFilter()
Constructor for TimeSeriesClassFilter.
Method

ComputeLags
virtual public double[,] ComputeLags(int nLags, double[,] x)
Lags time series data to a format used for input to a neural network.

Lags must be greater than 0.
It is assumed that x is sorted in descending chronological order.

nLags  An int containing the requested number of lags.
x  A double matrix, nObs by nVar, containing the time series data to be lagged.

Returns — A double matrix with (nObs-nLags) rows and (nVar*(nLags+1)) columns. The columns 0 through (nVar-1) contain the columns of x. The next nVar columns contain the first lag of the columns in x, etc.

Description

Class TimeSeriesFilter can be used to operate on a data matrix and lags every column to form a new data matrix. Using the method
Imsl.DataMining.Neural.TimeSeriesFilter.ComputeLags(System.Int32,System.Double[,]) (p. 739), each column of the input matrix, x, is transformed into (nLags+1) columns by creating a column for lags = 0, 1, ..., nLags.

The output data array, z, can be symbolically represented as:

\[ z = [x[0] : x[1] : x[2] : \ldots : x[nLags - 1]], \]

where x[i] is a lagged column of the incoming data matrix, x.

Consider, an example in which x has five rows and two columns with all variables continuous input attributes. Using nObs and nVar to represent the number of rows and columns in x, let

\[ x = \begin{bmatrix}
1 & 6 \\
2 & 7 \\
3 & 8 \\
4 & 9 \\
5 & 10
\end{bmatrix} \]

If nLags=1, then the number of columns in z[,] is nVar*(nLags+1) = 2*2 = 4, and the number of rows is (nObs-nLags) = 5-1 = 4:

\[ z = \begin{bmatrix}
1 & 6 & 2 & 7 \\
2 & 7 & 3 & 8 \\
3 & 8 & 4 & 9 \\
4 & 9 & 5 & 10
\end{bmatrix} \]
If nLags = 2, then the number of rows in z will be \( (nObs - nLags) = (5-2) = 3 \) and the number of columns will be \( nVar \times (nLags+1) = 2 \times 3 = 6 \):

\[
z = \begin{bmatrix} 1 & 6 & 2 & 7 & 3 & 8 \\
2 & 7 & 3 & 8 & 4 & 9 \\
3 & 8 & 4 & 9 & 5 & 10 \end{bmatrix}
\]

**Example: TimeSeriesFilter**

In this example a matrix with 5 rows and 2 columns is lagged twice. This produces a two-dimensional matrix with 5 rows, but \( 2 \times 3 = 6 \) columns. The first two columns correspond to lag=0, which just places the original data into these columns. The 3rd and 4th columns contain the first lags of the original 2 columns and the 5th and 6th columns contain the second lags.

```csharp
using System;
using Imsl.Stat;
using Imsl.Math;
using Imsl.DataMining.Neural;

public class TimeSeriesFilterEx1
{
    [STAThread]
    public static void Main(System.String[] args)
    {
        TimeSeriesFilter filter = new TimeSeriesFilter();
        int nLag = 2;
        double[,] x = {{1, 6}, {2, 7}, {3, 8}, {4, 9}, {5, 10}};
        double[,] z = filter.ComputeLags(nLag, x);
        // Print result without row/column labels.
        PrintMatrix pm = new PrintMatrix();
        PrintMatrixFormat mf;
        mf = new PrintMatrixFormat();
        mf.SetNoRowLabels();
        mf.SetNoColumnLabels();
        pm.SetTitle("Lagged data");
        pm.Print(mf, z);
    }
}
```

**Output**

```
Lagged data
1 6 2 7 3 8
2 7 3 8 4 9
3 8 4 9 5 10
```
TimeSeriesClassFilter Class

Converts time series data contained within nominal categories to a lagged format for processing by a neural network. Lagging is done within the nominal categories associated with the time series.

public class Imsl.DataMining.Neural.TimeSeriesClassFilter

Constructor

TimeSeriesClassFilter
    public TimeSeriesClassFilter(int nClasses)
    Constructor for TimeSeriesClassFilter.

    nClasses An int specifying the number of nominal categories associated with the time series.

Method

ComputeLags
    virtual public double[,] ComputeLags(int[] lags, int[] iClass, double[] x)
    Computes lags of an array sorted first by class designations and then descending chronological order.
    Every lag must be non-negative.
    The i-th element of iClass is equal to the class associated with the i-th element of x.
    iClass and x must be the same length.
    x is assumed to be sorted first by class designations and then descending chronological order; i.e., most recent observations appear first within a class.

    lags An int array containing the requested lags.
    iClass An int array containing class number associated with each element of x, sorted in ascending order.
    x A double array containing the time series data to be lagged.

    Returns — A double matrix containing the lagged data. The i-th column of this array is the lagged values of x for a lag equal to lags[i]. The number of rows is equal to the length of x.

Description

Class TimeSeriesClassFilter can be used with a data array, x to compute a new data array, z[,], containing lagged columns of x.
When using the method
(p. 741), the output array, \( z[\cdot] \) of lagged columns, can be symbolically represented as:

\[
\]

where \( x[i] \) is a lagged column of the incoming data array \( x \), and \( nLags \) is the number of computed lags. The lag associated with \( x[i] \) is equal to the value in \( \text{lags}[i] \), and lagging is done within the nominal categories given in \( iClass \). This requires the time series data in \( x[\cdot] \) be sorted in time order within each category \( iClass \).

Consider an example in which the number of observations in \( x[\cdot] \) is 10. There are two lags requested in \( \text{lags} \). If

\[
x^T = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\},
\]

\[
iClass^T = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1\},
\]

and

\[
\text{lag}^T = \{0, 2\}
\]

then, all the time series data fall into a single category, i.e. \( nClasses = 1 \), and \( z \) would contain 2 columns and 10 rows. The first column reproduces the values in \( x[\cdot] \) because \( \text{lags}[0] = 0 \), and the second column is the 2nd lag because \( \text{lags}[1] = 2 \).

\[
z = \begin{bmatrix}
1 & 3 \\
2 & 4 \\
3 & 5 \\
4 & 6 \\
5 & 7 \\
6 & 8 \\
7 & 9 \\
8 & 10 \\
9 & NaN \\
10 & NaN
\end{bmatrix}
\]

On the other hand, if the data were organized into two classes with

\[
iClass^T = \{1, 1, 1, 1, 2, 2, 2, 2, 2\},
\]

then \( nClasses \) is 2, and \( z \) is still a 2 by 10 matrix, but with the following values:

\[
z = \begin{bmatrix}
1 & 3 \\
2 & 4 \\
3 & 5 \\
4 & NaN \\
5 & NaN \\
6 & 8 \\
7 & 9 \\
8 & 10 \\
9 & NaN \\
10 & NaN
\end{bmatrix}
\]
The first 5 rows of $z$ are the lagged columns for the first category, and the last five are the lagged columns for the second category.

**Example: TimeSeriesClassFilter**

For illustration purposes, the time series in this example consists of the integers 1, 2, ..., 10, organized into two classes. Of course, it is assumed that these data are sorted in chronologically descending order. That is for each class, the first number is the latest value and the last number in that class is the earliest.

The values 1-4 are in class 1, and the values 5-10 are in class 2. These values represent two separate time series, one for each class. If you were to list them in chronologically ascending order, starting with time=$T_0$, the values would be:

Class 1: $T_0=4$, $T_1=3$, $T_2=2$, $T_3=1$
Class 2: $T_0=10$, $T_1=9$, $T_2=8$, $T_3=7$, $T_4=6$, $T_5=5$

This example requests lag calculations for lags 0, 1, 2, 3. For lag=0, no lagging is performed. For lag=1, the value at time = $t$ replaced with the value at time = $t-1$, the previous value in that class. If $t-1 < 0$, then a missing value is placed in that position.

For example, the first lag of a time series at time=$t$ are the values at time=$t-1$. For the time series values of Class 1 (lag=1), these values are:

Class 1, lag 1: $T_0=\text{NaN}$, $T_1=4$, $T_2=3$, $T_3=2$

The second lag for time=$t$ consists of the values at time=$t-2$:

Class 1, lag 2: $T_0=\text{NaN}$, $T_1=\text{NaN}$, $T_2=4$, $T_3=3$

Notice that the second lag now has two missing observations. In general, lag=n will have n missing values. In some cases this can result in all missing values for classes with few observations. A class will have all missing values in any of its lag columns that have a lag value larger than or equal to the number of observations in that class.

```csharp
using System;
using Imsl.Stat;
using Imsl.Math;
using Imsl.DataMining.Neural;

public class TimeSeriesClassFilterEx1
{
    private static int nClasses = 2;
    private static int nObs = 10;
    private static int nLags = 4;
    [STAThread]
    public static void Main(System.String[] args)
    {
        double[] x = new double[] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        double[] time = new double[] {3, 2, 1, 0, 5, 4, 3, 2, 1, 0};
    }
}
```
// Filter Classified Time Series Data
TimeSeriesClassFilter filter = new TimeSeriesClassFilter(nClasses);
double[,] y = filter.ComputeLags(lag, iClass, x);
double[,] z = new double[nObs, (nLags + 2)];
// for (int i = 0; i < nObs; i++)
// {
//  z[i] = new double[nLags + 2];
// }
for (int i = 0; i < nObs; i++)
{
    z[i, 0] = (double) iClass[i];
    z[i, 1] = time[i];
    for (int j = 0; j < nLags; j++)
    {
        z[i, j + 2] = y[i, j];
    }
}
// Print result without row/column labels.
PrintMatrix pm = new PrintMatrix();
PrintMatrixFormat mf;
mf = new PrintMatrixFormat();
mf.SetNoRowLabels();
mf.SetColumnLabels(colLabels);
pm.SetTitle("Lagged data");
pm.Print(mf, z);
}
Example: Neural Network Application

This application illustrates one common approach to time series prediction using a neural network. In this case, the output target for this network is a single time series. In general, the inputs to this network consist of lagged values of the time series together with other concomitant variables, both continuous and categorical. In this application, however, only the first three lags of the time series are used as network inputs.

The objective is to train a neural network for forecasting the series $Y_t, t = 0, 1, 2, \ldots$, from the first three lags of $Y_t$, i.e.

$$Y_t = f(Y_{t-1}, Y_{t-2}, Y_{t-3})$$

Since this series consists of data from several company departments, lagging of the series must be done within departments. This creates many missing values. The original data contains 118,519 training patterns. After lagging, 16,507 are identified as missing and are removed, leaving a total of 102,012 usable training patterns. Missing values are denoted using a number not in the training patterns, the value -9,999,999,999.0.

The structure of the network consists of three input nodes and two layers, with three perceptrons in the hidden layer and one in the output layer. The following figure depicts this structure:

![Diagram of a 2-layer Feed Forward Neural Network](image)

**Figure 9. An example 2-layer Feed Forward Neural Network**

There are a total of 16 weights in this network, including the 4 bias weights. All perceptrons in the hidden layer use logistic activation, and the output perceptron uses linear activation. Because of the large number of training patterns, the Activation.LogisticTable activation...
function is used instead of Activation.Logistic. Activation.LogisticTable uses a table lookup for calculating the logistic activation function, which significantly reduces training time. However, these are not completely interchangeable. If a network is trained using Activation.LogisticTable, then it is important to use the same activation function for forecasting.

All input nodes are linked to every perceptron in the hidden layer, which are in turn linked to the output perceptron. Then all inputs and the output target are scaled using the ScaleFilter class to ensure that all input values and outputs are in the range \([0, 1]\). This requires forecasts to be unscaled using the Decode() method of the ScaleFilter class.

Training is conducted using the epoch trainer. This trainer allows users to customize training into two stages. Typically this is necessary when training using a large number of training patterns. Stage I training uses randomly selected subsets of training patterns to search for network solutions. Stage II training is optional, and uses the entire set of training patterns. For larger sets of training patterns, training could take many hours, or even days. In that case, Stage II training might be bypassed.

In this example, Stage I training is conducted using the quasi-Newton trainer applied to 20 epochs, each consisting of 5,000 randomly selected observations. Stage II training also uses the quasi-Newton trainer.

The training patterns are contained in two data files: continuous.txt and output.txt. The formats of these files are identical. The first line of the file contains the number of columns or variables in that file. The second contains a line of tab-delimited integer values. These are the column indices associated with the incoming data. The remaining lines contain tab-delimited, floating point values, one for each of the incoming variables.

For example, the first four lines of the continuous.txt file consists of the following lines:

3
1 2 3
0 0 0
0 0 0

There are 3 continuous input variables which are numbered, or labeled, as 1, 2, and 3.

**Source Code**

```java
using System;
using Imsl.DataMining.Neural;
using Imsl.Math;
using System.Runtime.Serialization;
//****************************************************************************
// NeuralNetworkEx1.java *
// Two Layer Feed-Forward Network Complete Example for Simple Time Series *
//****************************************************************************
// Synopsis: This example illustrates how to use a Feed-Forward Neural *
// Network to forecast time series data. The network target is a *
```
time series and the three inputs are the 1st, 2nd, and 3rd lag for the target series.

Activation: Logistic_Table in Hidden Layer, Linear in Output Layer

Trainer: Epoch Trainer: Stage I - Quasi-Newton, Stage II - Quasi-Newton

Inputs: Lags 1-3 of the time series

Output: A Time Series sorted chronologically in descending order, i.e., the most recent observations occur before the earliest, within each department

ccakSerializable]
public class NeuralNetworkEx1 : System.Runtime.Serialization.ISerializable
{

private static System.String QuasiNewton = "quasi-newton";
private static System.String LeastSquares = "least-squares";

private static int nObs = 118519; // number of training patterns
private static int nInputs = 3; // four inputs
private static int nContinuous = 3; // one continuous input attribute
private static int nOutputs = 1; // one continuous output
private static int nPerceptrons = 3; // perceptrons in hidden layer
private static int[] perceptrons = new int[]{3}; // number of perceptrons in each hidden layer

// PERCEPTRON ACTIVATION
private static IActivation hiddenLayerActivation;
private static IActivation outputLayerActivation;

private static bool trace = true; //trainer logging
private static int nEpochs = 20; //number of epochs
private static int epochSize = 5000; //samples per epoch

private static int stage1Iterations = 5000; //max. iterations
private static double stage1StepTolerance = 1e-09; //step tolerance
private static double stage1RelativeTolerance = 1e-11; //rel. tolerance

private static int stage2Iterations = 5000; //max. iterations
private static double stage2StepTolerance = 1e-09; //step tolerance
private static double stage2RelativeTolerance = 1e-11; //rel. tolerance

// FILE NAMES AND FILE READER DEFINITIONS

private static System.IO.StreamReader contFileInputStream;
private static System.IO.StreamReader outputFileInputStream;

private static System.String trainingLogFile = "NeuralNetworkEx1.log";
private static System.String networkFileName = "NeuralNetworkEx1.ser";
private static System.String trainerFileName = "NeuralNetworkTrainerEx1.ser";

// READERS

private static System.IO.StreamReader contFileInputStream;
private static System.IO.StreamReader outputFileInputStream;

private static System.String trainingLogFile = "NeuralNetworkEx1.log";
private static System.String networkFileName = "NeuralNetworkEx1.ser";
private static System.String trainerFileName = "NeuralNetworkTrainerEx1.ser";
private static System.String xDataFileName = "NeuralNetworkxDataEx1.ser";
private static System.String yDataFileName = "NeuralNetworkyDataEx1.ser";

// INPUT FILES
// Continuous training attributes file. File contains Lags 1-3 of series
private static System.String contFile = "continuous.txt";
// Continuous network targets file. File contains the original series
private static System.String outputFile = "output.txt";

// *****************************************************************************
// Data Preprocessing Settings
// *****************************************************************************
private static double lowerDataLimit = -105000; // lower scale limit
private static double upperDataLimit = 25000000; // upper scale limit
// indicator

private static int startTime;

// *****************************************************************************
// Time Parameters for Tracking Training Time
// *****************************************************************************

private static System.String errorMsg = "";

private static System.String errorMsg0 = "--> Network Training";
private static System.String errorMsg1 = "--> The last global step failed to locate a lower point than the current solution. The current solution may be an approximate solution and no more accuracy is possible, or the step tolerance may be too large.";
private static System.String errorMsg2 = "--> Relative function convergence; both both the actual and predicted relative reductions in the error function are less than or equal to the relative function convergence tolerance.";
private static System.String errorMsg3 = "--> Scaled step tolerance satisfied; the current solution may be a local minimum of the error function, or the algorithm is making very slow progress and is not near a solution, or the step tolerance is too big.";
private static System.String errorMsg4 = "--> Quasi-Newton Trainer threw a MinUnconMultiVar.FalseConvergenceException.";
private static System.String errorMsg5 = "--> Quasi-Newton Trainer threw a MinUnconMultiVar.MaxIterationsException.";
private static System.String errorMsg6 = "--> Quasi-Newton Trainer threw a MinUnconMultiVar.UnboundedBelowException.";

// MAIN

[STAThread]
public static void Main(System.String[] args)
{
  double[] weight; // Network weights
  double[] gradient; // Network gradient after training
  double[,] xData; // Training Patterns Input Attributes
  double[,] yData; // Training Targets Output Attributes
  double[,] contAtt; // A 2D matrix for the continuous training attributes
  double[,] outs; // A matrix containing the training output targets
  int i, j, m = 0; // Array indices
  int nWeights = 0; // Number of network weights
  int nCol = 0; // Number of data columns in input file
  int[] ignore; // Array of 0's and 1's (0=missing value)
  int[] cont_col, outs_col, isMissing = new int[] {0};
  //System.String inputLine = "", temp;
  //System.String[] dataElement;
  // *****************************************************
  // Initialize timers
  // ****************************************************
NeuralNetworkEx1.startTime =
    DateTime.Now.Hour * 60 * 60 * 1000 +
    DateTime.Now.Minute * 60 * 1000 +
    DateTime.Now.Second * 1000 +
    DateTime.Now.Millisecond;


// *************************************************************************
// Read continuous attribute data *
// *************************************************************************
// Initialize ignore[] for identifying missing observations
ignore = new int[nObs];
isMissing = new int[1];
openInputFiles();

nCol = readFirstLine(contFileInputStream);

nContinuous = nCol;
System.Console.Out.WriteLine("--> Number of continuous variables: " + nContinuous);

if (nContinuous > 0)
{
    // contFile contains continuous attribute data
    contAtt = new double[nObs, nContinuous];
    double[] _contAttRow = new double[nContinuous];
    // for (int i2 = 0; i2 < nObs; i2++)
    // {
    //     contAtt[i2] = new double[nContinuous];
    // }
    // cont_col = readColumnLabels(contFileInputStream, nContinuous);
    for (i = 0; i < nObs; i++)
    {
        isMissing[0] = -1;
        _contAttRow = readDataLine(contFileInputStream, nContinuous, isMissing);
        for (int jj=0; jj < nContinuous; jj++)
        {
            contAtt[i,jj] = _contAttRow[jj];
        }
        ignore[i] = isMissing[0];
        if (isMissing[0] >= 0)
            m++;
    }
}
else
{
    nContinuous = 0;
    contAtt = new double[1,1];
    // for (int i3 = 0; i3 < 1; i3++)
    // {
    //     contAtt[i3] = new double[1];
    // }
    contAtt[0,0] = 0;
nCol = readFirstLine(outputFileInputStream);

System.Console.Out.WriteLine("--> Number of output variables: " + nOutputs);
outs = new double[nObs, nOutputs];
double[] _outsRow = new double[nOutputs];
// for (int i4 = 0; i4 < nObs; i4++)
// {
//   outs[i4] = new double[nOutputs];
// }
// Read numeric labels for continuous input attributes
outs_col = readColumnLabels(outputFileInputStream, nOutputs);

m = 0;
for (i = 0; i < nObs; i++)
{
  isMissing[0] = ignore[i];
  _outsRow = readDataLine(outputFileInputStream, nOutputs, isMissing);
  for (int jj = 0; jj < nOutputs; jj++)
  {
    outs[i, jj] = _outsRow[jj];
  }
  ignore[i] = isMissing[0];
  if (isMissing[0] >= 0)
    m++;
}
System.Console.Out.WriteLine("--> Number of Missing Observations: " + m);
closeFile(outputFileInputStream);

m = removeMissingData(nObs, nContinuous, ignore, contAtt);
m = removeMissingData(nObs, nOutputs, ignore, outs);

System.Console.Out.WriteLine("--> Total Number of Training Patterns: " + nObs);
nObs = nObs - m;
System.Console.Out.WriteLine("--> Number of Usable Training Patterns: " + nObs);

/*
* Setup Method and Bounds for Scale Filter
* ScaleFilter scaleFilter = new ScaleFilter(ScaleFilter.ScalingMethod.Bounded);
scaleFilter.SetBounds(lowerDataLimit, upperDataLimit, 0, 1);
*/
System.Console.Out.WriteLine("--> Starting Preprocessing of Training Patterns");
xData = new double[nObs, nContinuous];
// for (int i5 = 0; i5 < nObs; i5++)
// {
//   xData[i5] = new double[nContinuous];
// }
yData = new double[nObs, nOutputs];
// for (int i6 = 0; i6 < nObs; i6++)
// {
// {
// yData[i6] = new double[nOutputs];
// }
for (i = 0; i < nObs; i++)
{
for (j = 0; j < nContinuous; j++)
{
xData[i,j] = contAtt[i,j];
}
yData[i,0] = outs[i,0];
}
scaleFilter.Encode(0, xData);
scaleFilter.Encode(1, xData);
scaleFilter.Encode(2, xData);
scaleFilter.Encode(0, yData);
// *************************************************************************
// CREATE FEEDFORWARD NETWORK
*
// *************************************************************************
System.Console.Out.WriteLine("--> Creating Feed Forward Network Object");
FeedForwardNetwork network = new FeedForwardNetwork();
// setup input layer with number of inputs = nInputs = 3
network.InputLayer.CreateInputs(nInputs);
// create a hidden layer with nPerceptrons=3 perceptrons
network.CreateHiddenLayer().CreatePerceptrons(nPerceptrons);
// create output layer with nOutputs=1 output perceptron
network.OutputLayer.CreatePerceptrons(nOutputs);
// link all inputs and perceptrons to all perceptrons in the next layer
network.LinkAll();
// Get Network Perceptrons for Setting Their Activation Functions
Perceptron[] perceptrons = network.Perceptrons;
// Set all hidden layer perceptrons to logistic_table activation
for (i = 0; i < perceptrons.Length - 1; i++)
{
perceptrons[i].Activation = hiddenLayerActivation;
}
perceptrons[perceptrons.Length - 1].Activation = outputLayerActivation;
System.Console.Out.WriteLine("--> Feed Forward Network Created with 2 Layers");
// ****************************************************************************
// TRAIN NETWORK USING EPOCH TRAINER
*
// ****************************************************************************
ITrainer trainer = createTrainer(QuasiNewton, QuasiNewton);
startTime =
DateTime.Now.Hour * 60 * 60 * 1000 +
DateTime.Now.Minute * 60 * 1000 +
DateTime.Now.Second * 1000 +
DateTime.Now.Millisecond;
// Train Network
trainer.Train(network, xData, yData);
// Check Training Error Status
switch (trainer.ErrorStatus)
{
case 0: errorMsg = errorMsg0;
break;

Neural Nets

TimeSeriesClassFilter Class • 751


case 1:  errorMsg = errorMsg1;
    break;

case 2:  errorMsg = errorMsg2;
    break;

case 3:  errorMsg = errorMsg3;
    break;

case 4:  errorMsg = errorMsg4;
    break;

case 5:  errorMsg = errorMsg5;
    break;

case 6:  errorMsg = errorMsg6;
    break;

default:  errorMsg = "--> Unknown Error Status Returned from Trainer";
    break;

} 
System.Console.Out.WriteLine(errorMsg);

int currentTimeNow =
    DateTime.Now.Hour * 60 * 60 * 1000 +
    DateTime.Now.Minute * 60 * 1000 +
    DateTime.Now.Second * 1000 +
    DateTime.Now.Millisecond;

//UPGRADE_TODO: Method 'java.util.Date.toString' was converted to 'System.DateTime.ToString' which has a different behavior. 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="jlca1073"'
//UPGRADE_TODO: The equivalent in .NET for method 'java.util.Calendar.getTime' may return a different value. 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="jlca1043"'
//UPGRADE_ISSUE: Method 'java.util.Calendar.getTimeInMillis' was not converted. 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="jlca1000_javautilCalendargetTimeInMillis"'
double duration = (double) (currentTimeNow - startTime) / 1000.0;

// *************************************************************************
// DISPLAY TRAINING STATISTICS *
// *************************************************************************

double[] stats = network.ComputeStatistics(xData, yData);
// Display Network Errors
System.Console.Out.WriteLine("***************************************************************

//UPGRADE_WARNING: Narrowing conversions may produce unexpected results in C#. 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="jlca1042"
System.Console.Out.WriteLine("--> SSE: " + (float) stats[0]);
//UPGRADE_WARNING: Narrowing conversions may produce unexpected results in C#. 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="jlca1042"
System.Console.Out.WriteLine("--> RMS: " + (float) stats[1]);
//UPGRADE_WARNING: Narrowing conversions may produce unexpected results in C#. 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="jlca1042"
System.Console.Out.WriteLine("--> Largest Absolute Residual: " + (float) stats[4]);
System.Console.Out.WriteLine("***************************************************************");
// OBTAIN AND DISPLAY NETWORK WEIGHTS AND GRADIENTS *
// ****************************************************************************
// Get weights
weight = network.Weights;
// Get number of weights = number of gradients
nWeights = network.NumberOfWeights;
// Obtain Gradient Vector
gradient = trainer.ErrorGradient;
// Print Network Weights and Gradients
System.Console.Out.WriteLine("");
System.Console.Out.WriteLine("***************************");
double[,] printMatrix = new double[nWeights,2];
// for (int i7 = 0; i7 < nWeights; i7++)
// {
// printMatrix[i7] = new double[2];
// }
for (i = 0; i < nWeights; i++)
{
    printMatrix[i,0] = weight[i];
    printMatrix[i,1] = gradient[i];
}
// Print result without row/column labels.
System.String[] colLabels = new System.String[]{"Weight", "Gradient");
PrintMatrix pm = new PrintMatrix();
PrintMatrixFormat mf;
mf = new PrintMatrixFormat();
mf.SetNoRowLabels();
mf.SetColumnLabels(colLabels);
pm.SetTitle("Weights and Gradients");
pm.Print(mf, printMatrix);
System.Console.Out.WriteLine("***************************");
// ****************************************************************************
// SAVE THE TRAINED NETWORK BY SAVING THE SERIALIZED NETWORK OBJECT *
// ****************************************************************************
System.Console.Out.WriteLine("n--"> Saving Trained Network into " + networkFileName);
write(network, networkFileName);
System.Console.Out.WriteLine("n--"> Saving Network Trainer into " + trainerFileName);
write(trainer, trainerFileName);
System.Console.Out.WriteLine("n--"> Saving xData into " + xDataFileName);
write(xData, xDataFileName);
System.Console.Out.WriteLine("n--"> Saving yData into " + yDataFileName);
write(yData, yDataFileName);
}
// ****************************************************************************
// OPEN DATA FILES *
// ****************************************************************************
static public void openInputFiles()
{
    try
    {
        // Continuous Input Attributes
        System.IO.Stream contInputStream = new System.IO.FileStream(contFileName, System.IO.FileMode.Open, System.IO FileAccess.Read);
    }
    catch (Exception e)
    {
        System.Console.Out.WriteLine("Error reading continuous input attributes: " + e.Message);
    }
}
contFileInputStream = new System.IO.StreamReader(contInputStream).BaseStream, System.Text.Encoding.UTF7);

// Continuous Output Targets
System.IO.Stream outputInputStream = new System.IO.FileStream(outputFileName, System.IO.FileMode.Open, System.IO.FileAccess.Read);

outputFileInputStream = new System.IO.StreamReader(outputInputStream).BaseStream, System.Text.Encoding.UTF7);

} catch (System.Exception e)
{
    //UPGRADE_TODO: The equivalent in .NET for method 'java.lang.Throwable.toString' may return a different value. 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="jlca1043"'
System.Console.Out.WriteLine("--ERROR: " + e);
System.Environment.Exit(0);
}

// *****************************************************************************
// READ FIRST LINE OF DATA FILE AND RETURN NUMBER OF COLUMNS IN FILE *
// *****************************************************************************
static public int readFirstLine(System.IO.StreamReader inputFile)
{
    System.String inputLine = "", temp;
    int nCol = 0;
    try
    {
        temp = inputFile.ReadLine();
        inputLine = temp.Trim();
        nCol = System.Int32.Parse(inputLine);
    }
    catch (System.Exception e)
    {
        //UPGRADE_TODO: The equivalent in .NET for method 'java.lang.Throwable.toString' may return a different value. 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="jlca1043"'
        System.Console.Out.WriteLine("--ERROR READING 1st LINE OF File " + e);
        System.Environment.Exit(0);
    }
    return nCol;
}

// *****************************************************************************
// READ COLUMN LABELS (2ND LINE IN FILE) *
// *****************************************************************************
static public int[] readColumnLabels(System.IO.StreamReader inputFile, int nCol)
{
    int[] contCol = new int[nCol];
    System.String inputLine = "", temp;
    System.String[] dataElement;
    // Read numeric labels for continuous input attributes
    try
    {
        temp = inputFile.ReadLine();
        inputLine = temp.Trim();
    }
    catch (System.Exception e)
    {
        //UPGRADE_TODO: The equivalent in .NET for method 'java.lang.Throwable.toString' may return a different value. 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="jlca1043"'
        System.Console.Out.WriteLine("--ERROR READING 2nd LINE OF FILE: " + e);
        System.Environment.Exit(0);
    }
    dataElement = inputLine.Split(new Char[] {' '});
    for (int i = 0; i < nCol; i++)
{  
  contCol[i] = System.Int32.Parse(dataElement[i]);
}
return contCol;

// ***************************************************************************
// READ DATA ROW
// ***************************************************************************
static public double[] readDataLine(System.IO.StreamReader inputFile, int nCol, int[] isMissing)
{
  double missingValueIndicator = -9999999999.0;
  double[] dataLine = new double[nCol];
  double[] contCol = new double[nCol];
  System.String inputLine = "", temp;
  System.String[] dataElement;
  try
  {
    temp = inputFile.ReadLine();
    inputLine = temp.Trim();
  }
  catch (System.Exception e)
  {
    //UPGRADE_TODO: The equivalent in .NET for method 'java.lang.Throwable.toString' may return a different value. 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="jlca1043"'
    System.Console.Out.WriteLine("-->ERROR READING LINE: " + e);
    System.Environment.Exit(0);
  }
  dataElement = inputLine.Split(new Char[] {' '});
  for (int j = 0; j < nCol; j++)
  {
    dataLine[j] = System.Double.Parse(dataElement[j]);
    if (dataLine[j] == missingValueIndicator)
      isMissing[0] = 1;
  }
  return dataLine;
}

// ***************************************************************************
// CLOSE FILE
// ***************************************************************************
static public void closeFile(System.IO.StreamReader inputFile)
{
  try
  {
    inputFile.Close();
  }
  catch (System.Exception e)
  {
    //UPGRADE_TODO: The equivalent in .NET for method 'java.lang.Throwable.toString' may return a different value. 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="jlca1043"'
    System.Console.Out.WriteLine("ERROR: Unable to close file: " + e);
    System.Environment.Exit(0);
  }
}

// ***************************************************************************
// REMOVE MISSING DATA
// ***************************************************************************
// Now remove all missing data using the ignore[] array
// and recalculate the number of usable observations, nObs

Neural Nets

TimeSeriesClassFilter Class • 755
This method is inefficient, but it works. It removes one case at a time, starting from the bottom. As a case (row) is removed, the cases below are pushed up to take it's place.

```csharp
static public int removeMissingData(int nObs, int nCol, int[] ignore, double[,] inputArray) {
    int m = 0;
    for (int i = nObs - 1; i >= 0; i--)
    {
        if (ignore[i] >= 0)
        {
            // the ith row contains a missing value
            // remove the ith row by shifting all rows below the
            // ith row up by one position, e.g. row i+1 -> row i
            m++;
            if (nCol > 0)
            {
                for (int j = i; j < nObs - m; j++)
                {
                    for (int k = 0; k < nCol; k++)
                    {
                        inputArray[j,k] = inputArray[j + 1,k];
                    }
                }
            }
        }
    }
    return m;
}
```

Create Stage I/Stage II Trainer

```csharp
static public ITrainer createTrainer(System.String s1, System.String s2)
{
    EpochTrainer epoch = null; // Epoch Trainer (returned by this method)
    QuasiNewtonTrainer stage1Trainer; // Stage I Quasi-Newton Trainer
    QuasiNewtonTrainer stage2Trainer; // Stage II Quasi-Newton Trainer
    LeastSquaresTrainer stage1LS; // Stage I Least Squares Trainer
    LeastSquaresTrainer stage2LS; // Stage II Least Squares Trainer
    int currentTimeNow; // Calendar time tracker

    // Create Epoch (Stage I/Stage II) trainer from above trainers.
    if (s1.Equals(QuasiNewton))
    {
        // Setup stage I quasi-newton trainer
        stage1Trainer = new QuasiNewtonTrainer();
        stage1Trainer.MaximumStepsize = maxStepSize;
        stage1Trainer.MaximumTrainingIterations = stage1Iterations;
        stage1Trainer.StepTolerance = stage1StepTolerance;
        if (s2.Equals(QuasiNewton))
        {
            stage2Trainer = new QuasiNewtonTrainer();
            stage2Trainer.MaximumStepsize = maxStepSize;
            stage2Trainer.MaximumTrainingIterations = stage2Iterations;
            epoch = new EpochTrainer(stage1Trainer, stage2Trainer);
        }
    }
    else
    {
        stage1LS = new LeastSquaresTrainer();
        stage1LS.MaximumStepsize = maxStepSize;
        stage1LS.MaximumTrainingIterations = stage1Iterations;
        stage2LS = new LeastSquaresTrainer();
        stage2LS.MaximumStepsize = maxStepSize;
        stage2LS.MaximumTrainingIterations = stage2Iterations;
        stage2LS.StepTolerance = stage2StepTolerance;
        epoch = new EpochTrainer(stage1LS, stage2LS);
    }
}```
else
{
    if (s2.Equals(LeastSquares))
    {
        stage2LS = new LeastSquaresTrainer();
        stage2LS.InitialTrustRegion = 1.0e-3;
        //stage2LS.setMaximumStepsize(maxStepSize);
        stage2LS.MaximumTrainingIterations = stage2Iterations;
        epoch = new EpochTrainer(stage1Trainer, stage2LS);
    }
    else
    {
        epoch = new EpochTrainer(stage1Trainer);
    }
}
else
{
    // Setup stage I least squares trainer
    stage1LS = new LeastSquaresTrainer();
    stage1LS.InitialTrustRegion = 1.0e-3;
    stage1LS.MaximumTrainingIterations = stage1Iterations;
    //stage1LS.setMaximumStepsize(maxStepSize);
    if (s2.Equals(QuasiNewton))
    {
        stage2Trainer = new QuasiNewtonTrainer();
        //stage2Trainer.setMaximumStepsize(maxStepSize);
        stage2Trainer.MaximumTrainingIterations = stage2Iterations;
        epoch = new EpochTrainer(stage1LS, stage2Trainer);
    }
    else
    {
        if (s2.Equals(LeastSquares))
        {
            stage2LS = new LeastSquaresTrainer();
            stage2LS.InitialTrustRegion = 1.0e-3;
            //stage2LS.setMaximumStepsize(maxStepSize);
            stage2LS.MaximumTrainingIterations = stage2Iterations;
            epoch = new EpochTrainer(stage1LS, stage2LS);
        }
        else
        {
            epoch = new EpochTrainer(stage1LS);
        }
    }
}
epoch.NumberOfEpochs = nEpochs;
epoch.EpochSize = epochSize;
epoch.Random = new Imsl.Stat.Random(1234567);
System.Console.Out.WriteLine(" --> Trainer: Stage I - " + s1 + " Stage II " + s2);
System.Console.Out.WriteLine(" --> Number of Epochs: " + nEpochs);
System.Console.Out.WriteLine(" --> Epoch Size: " + epochSize);
// Describe optimization setup for Stage I training
System.Console.Out.WriteLine(" --> Creating Stage I Trainer");
System.Console.Out.WriteLine(" --> Stage I Iterations: " + stage1Iterations);
System.Console.Out.WriteLine(" --> Stage I Step Tolerance: " + stage1StepTolerance);
System.Console.Out.WriteLine(" --> Stage I Relative Tolerance: " + stage1RelativeTolerance);
System.Console.Out.WriteLine(" --> Stage I Step Size: " + "DEFAULT");
System.Console.Out.WriteLine(" --> Stage I Trace: " + trace);
if (s2.Equals(QuasiNewton) || s2.Equals(LeastSquares))
{
    // Describe optimization setup for Stage II training
    System.Console.Out.WriteLine(" --> Creating Stage II Trainer");
    System.Console.Out.WriteLine(" --> Stage II Iterations: " + stage2Iterations);
    System.Console.Out.WriteLine(" --> Stage II Step Tolerance: " + stage2StepTolerance);
    System.Console.Out.WriteLine(" --> Stage II Relative Tolerance: " + stage2RelativeTolerance);
    System.Console.Out.WriteLine(" --> Stage II Step Size: " + "DEFAULT");
    System.Console.Out.WriteLine(" --> Stage II Trace: " + trace);
}
// if (trace)
// {
//     try
//     {
//         Handler handler = new FileHandler(trainingLogFile);
//         Logger logger = Logger.getLogger("Imsl.DataMining.Neural");
//         logger.setLevel(Level.FINEST);
//         logger.addHandler(handler);
//         handler.setFormatter(EpochTrainer.Formatter);
//         System.Console.Out.WriteLine(" --> Training Log Stored in " + trainingLogFile);
//     }
//     catch (System.Exception e)
//     {
//         SupportClass.WriteStackTrace(e, Console.Error);
//     }
// }
currentTimeNow =
    DateTime.Now.Hour * 60 * 60 * 1000 +
    DateTime.Now.Minute * 60 * 1000 +
    DateTime.Now.Second * 1000 +
    DateTime.Now.Millisecond;
//UPGRADE_TODO: Method 'java.util.Date.toString' was converted to 'System.DateTime.ToString' which has a different behavior. 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="jlca1073"'
//UPGRADE_TODO: The equivalent in .NET for method 'java.util.Calendar.getTime' may return a different value. 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="jlca1043"'
// Return Stage I/Stage II trainer
return epoch;
}

// ********************************************************************************
// WRITE SERIALIZED OBJECT TO A FILE
// ********************************************************************************
static public void write(System.Object obj, System.String filename)
{
    System.IO.FileStream fos = new System.IO.FileStream(filename, System.IO FileMode.Create);
    IFormatter oos = new BinaryFormatter();
    oos.Serialize(fos, obj);
    fos.Close();
}
static NeuralNetworkEx1()
hiddenLayerActivation = Imsl.DataMining.Neural.Activation.LogisticTable;
outputLayerActivation = Imsl.DataMining.Neural.Activation.Linear;
}
// ***************************************************************************

Output

--> Starting Data Preprocessing at: 44821683
--> Number of continuous variables: 3
--> Number of output variables: 1
--> Number of Missing Observations: 16507
--> Total Number of Training Patterns: 118519
--> Number of Usable Training Patterns: 102012
--> Starting Preprocessing of Training Patterns
--> Creating Feed Forward Network Object
--> Feed Forward Network Created with 2 Layers
--> Training Network using Epoch Trainer
  --> Creating Epoch Trainer
    --> Trainer: Stage I - quasi-newton Stage II quasi-newton
    --> Number of Epochs: 20
    --> Epoch Size: 5000
  --> Creating Stage I Trainer
    --> Stage I Iterations: 5000
    --> Stage I Step Tolerance: 1E-09
    --> Stage I Relative Tolerance: 1E-11
    --> Stage I Step Size: DEFAULT
    --> Stage I Trace: True
  --> Creating Stage II Trainer
    --> Stage II Iterations: 5000
    --> Stage II Step Tolerance: 1E-09
    --> Stage II Relative Tolerance: 1E-11
    --> Stage II Step Size: DEFAULT
    --> Stage II Trace: True
--> Starting Network Training at 45070408
--> The last global step failed to locate a lower point than the current error value. The current solution may be an approximate solution and no more accuracy is possible, or the step tolerance may be too large.
--> Network Training Completed at: 52311842
--> Training Time: 7241.434 seconds

***********************************************************************
--> SSE: 4.49772
--> RMS: 0.1423779
--> Laplacian Error: 103.4631
--> Scaled Laplacian Error: 0.1707173
--> Largest Absolute Residual: 0.4921748
***********************************************************************

--> Getting Network Weights and Gradients

--> Network Weights and Gradients:
***********************************************************************
Weights and Gradients

Weight Gradient
-248.426149158357 -9.50818419128144E-05
-4.01301691047852 -9.08459022567118E-07
248.602873209042 -2.84623837579401E-05
258.622104579914 -8.49451049786515E-05
0.125785905718184 -7.51083204612989E-07
-258.811023180973 -2.81816574426092E-05
-394.380943852438 -0.000125916731945308
-0.356726621727131 -5.25467092773031E-07
394.428311058654 -2.70798222353788E-05
422.855858784789 -1.40339989032276E-06
-1.01024906991467 -8.54119524733673E-07
422.85496914701 3.37315953950526E-08
91.0301743864326 -0.000555459860183764
0.67227924955327 -3.1195756142863E-06
-91.0431760187523 -0.000120208750794691
-422.186774012951 -1.36686903761536E-06

---

---> Saving Trained Network into NeuralNetworkEx1.ser
---> Saving Network Trainer into NeuralNetworkTrainerEx1.ser
---> Saving xData into NeuralNetworkxDataEx1.ser
---> Saving yData into NeuralNetworkyDataEx1.ser

Results

The above output indicates that the network successfully completed its training. The final sum of squared errors was 3.88, and the RMS (the scaled version of the sum of squared errors) was 0.12. All of the gradients at this solution are nearly zero, which is expected if network training found a local or global optima. Non-zero gradients usually indicate there was a problem with network training.

---

---> Starting Data Preprocessing at: 41269770
---> Number of continuous variables: 3
---> Number of output variables: 1
---> Number of Missing Observations: 16507
---> Total Number of Training Patterns: 118519
---> Number of Usable Training Patterns: 102012
---> Starting Preprocessing of Training Patterns
---> Creating Feed Forward Network Object
---> Feed Forward Network Created with 2 Layers
---> Training Network using Epoch Trainer
  ---> Creating Epoch Trainer
    ---> Trainer: Stage I - quasi-newton Stage II quasi-newton
    ---> Number of Epochs: 20
    ---> Epoch Size: 5000
    ---> Creating Stage I Trainer
      ---> Stage I Iterations: 5000
      ---> Stage I Step Tolerance: 1E-09
      ---> Stage I Relative Tolerance: 1E-11
      ---> Stage I Step Size: DEFAULT
--> Stage I Trace:  True
--> Creating Stage II Trainer
--> Stage II Iterations:  5000
--> Stage II Step Tolerance:  1E-09
--> Stage II Relative Tolerance:  1E-11
--> Stage II Step Size:  DEFAULT
--> Stage II Trace:  True

--> Starting Network Training at 41314606
--> The last global step failed to locate a lower point than the current error value. The current solution may be an approximate solution and no more accuracy is possible, or the step tolerance may be too large.

--> Network Training Completed at: 42818045
--> Training Time:  1503.439 seconds

**********************************************
--> SSE:  4.49772
--> RMS:  0.1423779
--> Laplacian Error:  103.4631
--> Scaled Laplacian Error:  0.1707173
--> Largest Absolute Residual:  0.4921748
**********************************************

--> Getting Network Weights and Gradients

--> Network Weights and Gradients:

Weights and Gradients

<table>
<thead>
<tr>
<th>Weight</th>
<th>Gradient</th>
</tr>
</thead>
<tbody>
<tr>
<td>-248.425149158357</td>
<td>-9.50818419128144E-05</td>
</tr>
<tr>
<td>-0.01301691047852</td>
<td>-9.08459022567118E-07</td>
</tr>
<tr>
<td>248.602873209042</td>
<td>-2.84623837579401E-05</td>
</tr>
<tr>
<td>258.622104579914</td>
<td>-8.49510497865156E-05</td>
</tr>
<tr>
<td>0.12578595718184</td>
<td>-7.51083204612989E-07</td>
</tr>
<tr>
<td>-258.811023180973</td>
<td>-2.81816574426092E-05</td>
</tr>
<tr>
<td>-394.380943852438</td>
<td>-0.00125916731945308</td>
</tr>
<tr>
<td>-0.356726621727131</td>
<td>-5.25467092773031E-07</td>
</tr>
<tr>
<td>394.428311058654</td>
<td>-2.70798222353788E-05</td>
</tr>
<tr>
<td>422.855858784789</td>
<td>-1.40339989032276E-06</td>
</tr>
<tr>
<td>-1.01024906891467</td>
<td>-8.54119524733673E-07</td>
</tr>
<tr>
<td>422.854960914701</td>
<td>3.37315953950526E-08</td>
</tr>
<tr>
<td>91.0301743864326</td>
<td>-0.000555459860183764</td>
</tr>
<tr>
<td>0.67279284955327</td>
<td>-3.11957565142863E-06</td>
</tr>
<tr>
<td>-91.0431760187523</td>
<td>-0.000120208750794691</td>
</tr>
<tr>
<td>-422.186774012951</td>
<td>-1.36686903761535E-06</td>
</tr>
</tbody>
</table>

**********************************************

--> Saving Trained Network into NeuralNetworkEx1.ser
--> Saving Network Trainer into NeuralNetworkTrainerEx1.ser
--> Saving xData into NeuralNetworkxDataEx1.ser
--> Saving yData into NeuralNetworkyDataEx1.ser
Links to Input Data Files Used in this Example and the Training Log:

Network Class

Neural network base class.

public class Imsl.DataMining.Neural.Network

Properties

InputLayer

abstract public Imsl.DataMining.Neural.InputLayer InputLayer {get; }
The InputLayer object. The Network InputLayer.

Links

abstract public Imsl.DataMining.Neural.Link[] Links {get; }
An array containing the Link objects in the Network. An array of Links associated with this Network.

NumberOfInputs

abstract public int NumberOfInputs {get; }
The number of Network inputs. An int which contains the number of inputs.

NumberOfLinks

abstract public int NumberOfLinks {get; }
The number of Network Links among the nodes. An int which contains the number of Imsl.DataMining.Neural.Link (p. 709)s in the Network.

NumberOfOutputs

abstract public int NumberOfOutputs {get; }
The number of Network output Imsl.DataMining.Neural.Perceptron (p. 706)s. An int which contains the number of outputs.

NumberOfWeights

abstract public int NumberOfWeights {get; }
The number of Imsl.DataMining.Neural.Link.Weight (p. 709)s in the Network. An int which contains the number of Weights associated with this Network.

OutputLayer

abstract public Imsl.DataMining.Neural.OutputLayer OutputLayer {get; }
The OutputLayer. The Network OutputLayer.
Perceptrons

abstract public Imsl.DataMining.Neural.Perceptron[] Perceptrons {get; }
An array containing the Perceptrons in the Network. An array of Perceptrons
associated with this Network.

Weights

abstract public double[] Weights {get; set; }
The Imsl.DataMining.Neural.Link.Weight (p. 709)s. A double array containing the
Weights associated with Network Imsl.DataMining.Neural.Link (p. 709)s.

Constructor

Network

public Network()
Default constructor for Network.

Since this class is abstract, it cannot be instantiated directly; this constructor is used by
constructors in classes derived from Network.

Methods

ComputeStatistics

virtual public double[] ComputeStatistics(double[,] xData, double[,] yData)
Computes error statistics.

This is a static method that can be used to compute the statistics regardless of the
training class used to train the network.

Computes statistics related to the error. In this table, the observed values are $y_i$. The
forecasted values are $\hat{y}_i$. The mean observed value is $\bar{y} = \sum y_i / NC$, where $N$ is the
number of observations and $C$ is the number of classes per observation.

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SSE</td>
<td>$\frac{1}{2} \sum (y_i - \bar{y})^2$</td>
</tr>
<tr>
<td>1</td>
<td>RMS</td>
<td>$\frac{\sum (y_i - \bar{y})^2}{\sum (y_i - \hat{y}_i)^2}$</td>
</tr>
<tr>
<td>2</td>
<td>Laplacian</td>
<td>$\sum</td>
</tr>
<tr>
<td>3</td>
<td>Scaled Laplacian</td>
<td>$\sum</td>
</tr>
<tr>
<td>4</td>
<td>Max residual</td>
<td>$\max_i</td>
</tr>
</tbody>
</table>

xData  A double matrix containing the input values.

yData  A double array containing the observed values.

Returns — A double array containing the above described statistics.

CreateHiddenLayer

abstract public Imsl.DataMining.Neural.HiddenLayer CreateHiddenLayer()
Creates the next HiddenLayer in the Network.
Returns — The new HiddenLayer.

Forecast
abstract public double[] Forecast(double[] x)
Returns a forecast for each of the Network's outputs computed from the trained Network.

x  A double array of values with the same length and order as the training patterns used to train the Network.

Returns — A double array containing the forecasts for the output Imsl.DataMining.Neural.Perceptron (p. 706)s. Its length is equal to the number of output Perceptrons.

GetForecastGradient
abstract public double[] GetForecastGradient(double[] x)
Returns the first derivatives with respect to the Imsl.DataMining.Neural.Link.Weight (p. 709)s evaluated at x.

x  A double array which specifies the input values at which the gradient is to be evaluated. It must have the same length and order as the training patterns used to train the Network.

Returns — A double array containing the derivative values. The i-th entry in this array contains dN(xData, weights)/d weights[i]. Its length is equal to the number of Imsl.DataMining.Neural.Link.Weight (p. 709)s in the Network.

Example: Network

This example uses a network previously trained and serialized into four files to obtain information about the network and forecasts. Training was done using the code for the FeedForwardNetwork Example 1.

The network training targets were generated using the relationship:

\[ y = 10X_1 + 20X_2 + 30X_3 + 2.0X_4, \]

where \( X_1-X_3 \) are the three binary columns, corresponding to categories 1 to 3 of the nominal attribute, and \( X_4 \) is the scaled continuous attribute.

The structure of the network consists of four input nodes and two layers, with three perceptrons in the hidden layer and one in the output layer. The following figure illustrates this structure:
Figure 10. An example 2-layer Feed Forward Neural Network with 4 Inputs

All perceptrons were trained using a Linear Activation Function. Forecasts are generated for 9 conditions, corresponding to the following conditions:
- Nominal Class 1-3 with the Continuous Input Attribute = 0
- Nominal Class 1-3 with the Continuous Input Attribute = 5.0
- Nominal Class 1-3 with the Continuous Input Attribute = 10.0

Note that the network training statistics retrieved from the serialized network confirm that this is the same network used in the previous example. Obtaining these statistics requires retrieval of the training patterns which were serialized and stored into separate files. This information is not serialized with the network, nor with the trainer.

using System;
using Imsl.DataMining.Neural;
using System.Runtime.Serialization;
//*****************************************************************************/
// Two Layer Feed-Forward Network with 4 inputs: 1 categorical with 3 classes
// encoded using binary encoding and 1 continuous input, and 1 output
// target (continuous). There is a perfect linear relationship between
// the input and output variables:
//
// MODEL: \( Y = 10 \times X_1 + 20 \times X_2 + 30 \times X_3 + 2 \times X_4 \)
//
// Variables X1-X3 are the binary encoded nominal variable and X4 is the
// This example uses Linear Activation in both the hidden and output layers
// The network uses a 2-layer configuration, one hidden layer and one
// output layer. The hidden layer consists of 3 perceptrons. The output
// layer consists of a single output perceptron.
// The input from the continuous variable is scaled to [0,1] before training
// the network. Training is done using the Quasi-Newton Trainer.
// The network has a total of 19 weights.
// Since the network target is a linear combination of the network inputs, and
// since all perceptrons use linear activation, the network is able to forecast
// the every training target exactly. The largest residual is 2.78E-08.

[Serializable]
public class NetworkEx1
{
    // ******************************************************************************
    // MAIN
    // ******************************************************************************
    [STAThread]
    public static void Main(System.String[] args)
    {
        double[,] xData; // Input Attributes for Training Patterns
        double[,] yData; // Output Attributes for Training Patterns
        double[] weight; // network weights
        double[] gradient; // network gradient after training
        // Input Attributes for Forecasting
        double[,] x = {{1, 0, 0, 0.0}, {0, 1, 0, 0.0}, {0, 0, 1, 0.0},
                        {1, 0, 0, 5.0}, {0, 1, 0, 5.0}, {0, 0, 1, 5.0},
                        {1, 0, 0, 10.0}, {0, 1, 0, 10.0}, {0, 0, 1, 10.0}};
        double[] xTemp, y; // Temporary areas for storing forecasts
        int i, j; // loop counters
        // Names of Serialized Files
        System.String networkFileName = "FeedForwardNetworkEx1.ser"; // the network
        System.String trainerFileName = "FeedForwardTrainerEx1.ser"; // the trainer
        System.String xDataFileName = "FeedForwardxDataEx1.ser"; // xData
        System.String yDataFileName = "FeedForwardyDataEx1.ser"; // yData
        // READ THE TRAINED NETWORK FROM THE SERIALIZED NETWORK OBJECT
        System.Console.Out.WriteLine("-- Reading Trained Network from " + networkFileName);
        Network network = (Network) read(networkFileName);
        // READ THE SERIALIZED XDATA[,] AND YDATA[,] ARRAYS OF TRAINING PATTERNS.
        System.Console.Out.WriteLine("-- Reading Network Trainer from " + trainerFileName);
        ITrainer trainer = (ITrainer) read(trainerFileName);
        System.Console.Out.WriteLine("-- Reading xData from " + xDataFileName);
        xData = (double[,]) read(xDataFileName);
        System.Console.Out.WriteLine("-- Reading yData from " + yDataFileName);
        yData = (double[,]) read(yDataFileName);
    }
}

--- Network Class IMSL C# Numerical Library ---
double[] stats = network.ComputeStatistics(xData, yData);

// Display Network Errors
System.Console.Out.WriteLine("************************************************");
System.Console.Out.WriteLine("--> SSE: " + (float) stats[0]);
System.Console.Out.WriteLine("--> RMS: " + (float) stats[1]);
System.Console.Out.WriteLine("--> Largest Absolute Residual: " + (float) stats[4]);
System.Console.Out.WriteLine(" ****************************************************");

// OBTAIN AND DISPLAY NETWORK WEIGHTS AND GRADIENTS

float[] weight = network.getWeights();
int nWeights = network.numberOfWeights();

float[] gradient = trainer.getErrorGradient();


for (i = 0; i < nWeights; i++)
{
    System.Console.Out.WriteLine("w[" + i + "]=" + (float) weight[i] + " g[" + i + "]=" + (float) gradient[i]);
}

// OBTAIN AND DISPLAY FORECASTS FOR THE LAST 10 TRAINING TARGETS

int nInputs = network.numberOfInputs();
int nOutputs = network.numberOfOutputs();
xTemp = new float[nInputs]; // temporary x space for forecast inputs
y = new float[nOutputs]; // temporary y space for forecast output

System.Console.Out.WriteLine(" ");

for (i = 0; i < nWeights; i++)
{
    System.Console.Out.WriteLine("w[" + i + "]=" + (float) weight[i] + " g[" + i + "]=" + (float) gradient[i]);
}

// Obtain example forecasts for input attributes = x[]
// X1-X3 are binary encoded for one nominal variable with 3 classes
// X4 is a continuous input attribute ranging from 0-10. During
// training, X4 was scaled to [0,1] by dividing by 10.
for (i = 0; i < 9; i++)
{
    xTemp[i] = x[i];
    xTemp[nInputs - 1] = xTemp[nInputs - 1] / 10.0;
    y = network.Forecast(xTemp);
Output

--> Reading Trained Network from FeedForwardNetworkEx1.ser
--> Reading xData from FeedForwardxDataEx1.ser
--> Reading yData from FeedForwardyDataEx1.ser
--> Reading Network Trainer from FeedForwardTrainerEx1.ser
******************************************************************************
--> SSE: 1.013444E-15
--> RMS: 2.007463E-19
--> Laplacian Error: 3.005804E-07
--> Scaled Laplacian Error: 3.535235E-10
--> Largest Absolute Residual: 2.784275E-08
******************************************************************************

--> Getting Network Information

--> Network Weights and Gradients:
   w[0]=-1.491785 g[0]=-2.611079E-08
   w[1]=-1.491785 g[1]=-2.611079E-08
   w[8]=4.725622 g[8]=-5.273856E-08
   w[12]=1.072258 g[12]=-1.690978E-07
   w[14]=1.072258 g[14]=-1.690978E-07
   w[16]=3.850755 g[16]=-1.7029E-08

768 • Network Class

IMSL C# Numerical Library
\( w[17] = 3.850755 \) \( g[17] = -1.7029 \times 10^{-8} \)
\( w[18] = 2.411725 \) \( g[18] = -1.588144 \times 10^{-8} \)

\[
\begin{array}{cccc|cc|c}
\text{X1} & \text{X2} & \text{X3} & \text{X4} & y & \text{Forecast} \\
1 & 0 & 0 & 0 & 10 & 10 \\
0 & 1 & 0 & 0 & 20 & 20 \\
0 & 0 & 1 & 0 & 30 & 30 \\
1 & 1 & 0 & 0 & 20 & 20 \\
0 & 1 & 0 & 5 & 30 & 30 \\
0 & 0 & 1 & 5 & 40 & 40 \\
1 & 0 & 0 & 10 & 30 & 30 \\
0 & 1 & 0 & 10 & 40 & 40 \\
0 & 0 & 1 & 10 & 50 & 50 \\
\end{array}
\]
Chapter 24: Miscellaneous

Types

\texttt{class Warning} ................................................................. \texttt{771}
\texttt{class WarningObject} ......................................................... \texttt{772}
\texttt{class IMSLEException} ...................................................... \texttt{773}

Warning Class

Handles warning messages.

\texttt{public class Imsl.Warning}

Properties

\texttt{WarningObject}

\texttt{static public Imsl.WarningObject WarningObject \{get; set; \}}

The \texttt{WarningObject} allows warning errors to be handled in a more custom fashion. The current \texttt{WarningObject}.

\texttt{WarningObject} may be set to null, in which case error messages will be ignored.

\texttt{Writer}

\texttt{static public System.IO.TextWriter Writer \{get; set; \}}

The stream to which warning messages are to be written.

The input may be null, in which case warnings are not written.

Constructor

\texttt{Warning}
public Warning()
Initializes a new instance of the Imsl.Warning (p. 771) class.

Method

Print

static public void Print(Object source, string bundleName, string key, Object[] arg)
Issues a warning message.

Warning messages are stored as MessageFormat patterns in a ResourceBundle. This method retrieves the pattern from the bundle, formats the message with the supplied arguments, and prints the message to the warning stream.

source The Object that is the source of the warning.

bundleName A String which specifies the base name of the resource. The actual name is formed by appending ".ErrorMessages".

key A String which specifies the warning message in the resource.

arg A Object which specifies arguments used to format the message.

Description

This class maintains a single, private, WarningObject that actually displays the warning messages.

WarningObject Class

Handles warning messages.

public class Imsl.WarningObject

Property

Writer

public System.IO.TextWriter Writer {get; set; }
Reassigns the writer.

The new warning writer may be set to null, in which case warnings are not printed.
**Constructor**

```
 WARNING_OBJECT
   public WARNING_OBJECT()
   Handle warning messages.
```

**Method**

```
Print
   virtual public void Print(Object source, string baseName, string key, Object[] arg)
   Issue a warning message.
   Warning messages are stored as string format items in a resource. This method retrieves
   the format from the resource, formats the message with the supplied arguments, and
   prints the message to the warning stream.
   
   source   The Object that is the source of the warning.
   baseName   A String which specifies the base name of the resource. The actual name
              is formed by appending ".ErrorMessages".
   key   A String which specifies the warning message in the resource.
   arg   A Object which specifies arguments used to format the message.
```

---

**IMSLException Class**

Signals that a mathematical exception has occurred.

```csharp
public class Imsl.IMSLException : ApplicationException implements
```

**Properties**

**HResult**

```
   int HResult {get; set; }
   Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
```

**HelpLink**

```
   virtual public string HelpLink {get; set; }
   Gets or sets a link to the help file associated with this exception.
```

**InnerException**

```
   public System.Exception InnerException {get; }
   Gets the System.Exception instance that caused the current exception.
```
Message

    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source

    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace

    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.

TargetSite

    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

IMSLException

    IMSLException()
    Constructs an IMSLException with no detail message.
    A detail message is a String that describes this particular exception.

IMSLException

    IMSLException(string s)
    Constructs an IMSLException with the specified detail message.
    A detail message is a String that describes this particular exception.

    s    A String which specifies the detail message.

IMSLException

    IMSLException(string namespaceName, string key, Object[] arguments)
    Constructs an IMSLException with the specified detail message.
    The error message String is in a resource bundle, ErrorMessages.

    namespaceName    A String which specifies the namespace containing the
                      ErrorMessages resource bundle.

    key    A String which specifies the key of the error message in the resource bundle.

    arguments   An array of Objects containing arguments used within the error message
                string.

IMSLException

    IMSLException(string message, System.Exception exception)
    Constructs an IMSLException with the specified detail message.

    message    The error message that explains the reason for the exception.
**exception**  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**IMSLException**


Constructs an IMSLException with the serialized data.

info  The Object that holds the serialized object data.
context  The contextual information about the source or destination.

**Methods**

**GetBaseException**

virtual public System.Exception GetBaseException()

When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

**GetObjectData**

virtual public void


When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

**ToString**

override public string ToString()

Creates and returns a string representation of the current exception.
Chapter 25: Exceptions

Types

class BadInitialGuessException ........................................... 779
class BoundsInconsistentException .................................. 781
class ConstraintEvaluationException .................................. 783
class ConstraintsInconsistentException .............................. 785
class ConstraintsNotSatisfiedException .............................. 787
class DidNotConvergeException ......................................... 789
class EqualityConstraintsException ................................. 791
class FalseConvergenceException ..................................... 793
class IllConditionedException .......................................... 795
class InconsistentSystemException .................................... 797
class LimitingAccuracyException ...................................... 799
class LinearlyDependentGradientsException ....................... 801
class MaxIterationsException .......................................... 803
class MaxNumberStepsAllowedException ......................... 805
class NoAcceptableStepsizeException ............................... 807
class NotSPDException ................................................... 809
class NumericDifficultyException ..................................... 811
class ObjectiveEvaluationException ................................. 813
class PenaltyFunctionPointInfeasibleException .................... 815
class ProblemInfeasibleException .................................... 817
class ProblemUnboundedException .................................... 819
class QPInfeasibleException ........................................... 821
class SingularException .................................................. 823
class SingularMatrixException .......................................... 825
class TerminationCriteriaNotSatisfiedException .................. 827
class ToleranceTooSmallException ..................................... 829
class TooManyIterationsException .................................... 832
class UnboundedBelowException ....................................... 834
class VarBoundsInconsistentException .............................. 836
class WorkingSetSingularException .................................. 838
class AllDeletedException .............................................. 840
class AllMissingException ............................................... 842
class BadVarianceException ................................................................. 844
class ClassificationVariableException ........................................... 846
class ClassificationVariableLimitException .................................. 848
class ClassificationVariableValueException ................................. 850
class ClusterNoPointsException ..................................................... 852
class ConstrInconsistentException ................................................ 854
class CovarianceSingularException ................................................ 856
class CyclingIsOccurringException .............................................. 858
class DeleteObservationsException .............................................. 860
class DidNotConvergeException ..................................................... 862
class DiffObsDeletedException ...................................................... 864
class EigenvalueException ............................................................ 866
class EmptyGroupException ............................................................ 868
class EqConstrInconsistentException ............................................ 870
class IllConditionedException ....................................................... 872
class IncreaseErrRelException ...................................................... 874
class MatrixSingularException ....................................................... 876
class MoreObsDelThanEnteredException ....................................... 878
class NegativeFreqException ........................................................ 881
class NegativeWeightException ..................................................... 883
class NewInitialGuessException .................................................... 885
class NoConvergenceException ...................................................... 887
class NoDegreesOfFreedomException ............................................ 889
class NoVariationInputException ................................................... 916
class NoVectorXException ............................................................. 918
class NonPosVarianceException ..................................................... 895
class NonPosVarianceXYException ............................................... 897
class NonPositiveEigenvalueException ......................................... 899
class NoPositiveVarianceException .............................................. 901
class NotCDFException ............................................................... 903
class NotPositiveDefiniteException ............................................. 906
class NotPositiveSemiDefiniteException ..................................... 908
class NotSemiDefiniteException ................................................... 910
class NoVariablesEnteredException ............................................. 912
class NoVariablesException ......................................................... 914
class NoVariationInputException ................................................ 916
class NoVectorXException ............................................................ 918
class PooledCovarianceSingularException .................................... 920
class RankException ................................................................. 922
class ScaleFactorZeroException .................................................... 924
class SingularException ............................................................. 926
class SumOfWeightsNegException .................................................. 928
class TooManyCallsException ....................................................... 930
class TooManyFunctionEvaluationsException .................................. 932
class TooManyIterationsException ............................................... 934
class TooManyJacobianEvalException .......................................... 936
BadInitialGuessException Class

Penalty function point infeasible for original problem. Try new initial guess.

public class Imsl.Math.BadInitialGuessException : IMSLException implements

Properties

HRESULT

    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException

    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message

    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source

    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace

    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.
**Constructors**

BadInitialGuessException

```csharp
public BadInitialGuessException()
Penalty function point infeasible for original problem. Try new initial guess.
```

BadInitialGuessException

```csharp
public BadInitialGuessException(string message)
Penalty function point infeasible for original problem. Try new initial guess.

message The error message that explains the reason for the exception.
```

BadInitialGuessException

```csharp
public BadInitialGuessException(string message, System.Exception exception)
Penalty function point infeasible for original problem. Try new initial guess.

message The error message that explains the reason for the exception.

exception The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a
catch block that handles the inner exception.
```

BadInitialGuessException

```csharp
Penalty function point infeasible for original problem. Try new initial guess.

info The object that holds the serialized object data.

context The contextual information about the source or destination.
```

**Methods**

GetBaseException

```csharp
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.
```

GetObjectData

```csharp
virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the
```

ToString

```csharp
override public string ToString()
Creates and returns a string representation of the current exception.
```
BoundsInconsistentException Class

The bounds given are inconsistent.

```java
public class Imsl.Math.BoundsInconsistentException : IMSLException implements Properties

HResult
    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.
```

Constructors

```java
BoundsInconsistentException
    public BoundsInconsistentException(string nameVariable, string nameLowerBound, string nameUpperBound, int index, double lowerBound, double upperBound)
    The bounds given are inconsistent.
```
**nameVariable**  Name of the variable being bounded.

**nameLowerBound**  Name of the lower bound.

**nameUpperBound**  Name of the upper bound.

**index**  The index of the inconsistent bound.

**lowerBound**  Value of the lower bound.

**upperBound**  Value of the upper bound.

---

**BoundsInconsistentException**

public BoundsInconsistentException()

The bounds given are inconsistent.

---

**BoundsInconsistentException**

public BoundsInconsistentException(string message)

The bounds given are inconsistent.

  **message**  The error message that explains the reason for the exception.

---

**BoundsInconsistentException**

public BoundsInconsistentException(string s, System.Exception exception)

The bounds given are inconsistent.

  **s**  The error message that explains the reason for the exception.

  **exception**  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

---

**BoundsInconsistentException**


The bounds given are inconsistent.

  **info**  The object that holds the serialized object data.

  **context**  The contextual information about the source or destination.

---

**Methods**

**GetBaseException**

virtual public System.Exception GetBaseException()

When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

**GetObjectData**


When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.
ToString
override public string ToString()
    Creates and returns a string representation of the current exception.

ConstraintEvaluationException Class

Constraint evaluation returns an error with current point.

public class Imsl.Math.ConstraintEvaluationException : IMSLException
    implements

Properties

HResult
    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Exceptions
Constructors

ConstraintEvaluationException
   public ConstraintEvaluationException()
   Constraint evaluation returns an error with current point.

ConstraintEvaluationException
   public ConstraintEvaluationException(string message)
   Constraint evaluation returns an error with current point.
   message   The error message that explains the reason for the exception.

ConstraintEvaluationException
   public ConstraintEvaluationException(string s, System.Exception exception)
   Constraint evaluation returns an error with current point.
   s   The error message that explains the reason for the exception.
   exception   The exception that is the cause of the current exception. If the
                innerException parameter is not a null reference, the current exception is raised in a
                catch block that handles the inner exception.

ConstraintEvaluationException
   ConstraintEvaluationException(System.Runtime.Serialization.SerializationInfo
   Constraint evaluation returns an error with current point.
   info   The object that holds the serialized object data.
   context   The contextual information about the source or destination.

Methods

GetBaseException
   virtual public System.Exception GetBaseException()
   When overridden in a derived class, returns the System.Exception that is the root cause
   of one or more subsequent exceptions.

GetObjectData
   virtual public void
   GetObjectData(System.Runtime.Serialization.SerializationInfo info,
   When overridden in a derived class, sets the

ToString
   override public string ToString()
   Creates and returns a string representation of the current exception.
**ConstraintsInconsistentException Class**

The equality constraints are inconsistent.

```java
public class Imsl.Math.ConstraintsInconsistentException : IMSLException implements
```

**Properties**

**HResult**
```java
    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
```

**HelpLink**
```java
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.
```

**InnerException**
```java
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.
```

**Message**
```java
    virtual public string Message {get; }
    Gets a message that describes the current exception.
```

**Source**
```java
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.
```

**StackTrace**
```java
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.
```

**TargetSite**
```java
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.
```

**Constructors**

```java
ConstraintsInconsistentException
    public ConstraintsInconsistentException()
    The equality constraints are inconsistent.
```
ConstraintsInconsistentException

```csharp
public ConstraintsInconsistentException(string message)
    The equality constraints are inconsistent.

message  The error message that explains the reason for the exception.
```

```csharp
public ConstraintsInconsistentException(string s, System.Exception exception)
    The equality constraints are inconsistent.

s  The error message that explains the reason for the exception.

exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
```

```csharp
    The equality constraints are inconsistent.

info  The object that holds the serialized object data.

context  The contextual information about the source or destination.
```

**Methods**

**GetBaseException**

```csharp
virtual public System.Exception GetBaseException()
    When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.
```

**GetObjectData**

```csharp
    When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.
```

**ToString**

```csharp
override public string ToString()
    Creates and returns a string representation of the current exception.
```
ConstraintsNotSatisfiedException Class

No vector \( x \) satisfies all of the constraints.

```csharp
public class Imsl.Math.ConstraintsNotSatisfiedException : IMSLException
    implements
```

Properties

HResult
```csharp
    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
```

HelpLink
```csharp
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.
```

InnerException
```csharp
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.
```

Message
```csharp
    virtual public string Message {get; }
    Gets a message that describes the current exception.
```

Source
```csharp
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.
```

StackTrace
```csharp
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.
```

TargetSite
```csharp
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.
```

Constructors

```csharp
ConstraintsNotSatisfiedException
    public ConstraintsNotSatisfiedException()
    No vector \( x \) satisfies all of the constraints.
```
ConstraintsNotSatisfiedException

public ConstraintsNotSatisfiedException(string message)
No vector \( x \) satisfies all of the constraints.

message  The error message that explains the reason for the exception.

ConstraintsNotSatisfiedException

public ConstraintsNotSatisfiedException(string s, System.Exception exception)
No vector \( x \) satisfies all of the constraints.

s  The error message that explains the reason for the exception.

exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

ConstraintsNotSatisfiedException

No vector \( x \) satisfies all of the constraints.

info  The object that holds the serialized object data.

context  The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.
DidNotConvergeException Class

Maximum number of iterations exceeded.

```java
public class Imsl.Math.DidNotConvergeException : IMSLException implements
```

**Properties**

HResult

```java
int HResult {get; set; }
```

Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

```java
virtual public string HelpLink {get; set; }
```

Gets or sets a link to the help file associated with this exception.

InnerException

```java
public System.Exception InnerException {get; }
```

Gets the System.Exception instance that caused the current exception.

Message

```java
virtual public string Message {get; }
```

Gets a message that describes the current exception.

Source

```java
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

StackTrace

```java
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```java
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

**Constructors**

DidNotConvergeException

```java
public DidNotConvergeException()
```

Maximum number of iterations exceeded.
DidNotConvergeException

public DidNotConvergeException(int maximumNumberOfIterations)
Maximum number of iterations exceeded.

maximumNumberOfIterations Maximum number of iterations allowed exceeded argument.

DidNotConvergeException

public DidNotConvergeException(int info, int min)
Maximum number of iterations exceeded.

info First argument for SVD.DidNotConverge string.
min Second argument for SVD.DidNotConverge string.

DidNotConvergeException

public DidNotConvergeException(string message)
Maximum number of iterations exceeded.

message The error message that explains the reason for the exception.

DidNotConvergeException

public DidNotConvergeException(string message, int maximumNumberOfIterations)
Maximum number of iterations exceeded.

message The error message that explains the reason for the exception.
maximumNumberOfIterations Maximum number of iterations allowed.

DidNotConvergeException

public DidNotConvergeException(string s, System.Exception exception)
Maximum number of iterations exceeded.

s The error message that explains the reason for the exception.
exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a
  catch block that handles the inner exception.

DidNotConvergeException

Maximum number of iterations exceeded.

info The object that holds the serialized object data.
context The contextual information about the source or destination.
**Methods**

**GetBaseException**

```csharp
virtual public System.Exception GetBaseException()
```
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

**GetObjectData**

```csharp
virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
```
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

**ToString**

```csharp
override public string ToString()
```
Creates and returns a string representation of the current exception.

---

**EqualityConstraintsException Class**

The variables are determined by the equality constraints.

```csharp
public class Imsl.Math.EqualityConstraintsException : IMSLException implements
```

**Properties**

**HResult**

```csharp
int HResult {get; set; }
```
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

**HelpLink**

```csharp
virtual public string HelpLink {get; set; }
```
Gets or sets a link to the help file associated with this exception.

**InnerException**

```csharp
public System.Exception InnerException {get; }
```
Gets the System.Exception instance that caused the current exception.

**Message**

```csharp
virtual public string Message {get; }
```
Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

EqualityConstraintsException
    public EqualityConstraintsException()
    The variables are determined by the equality constraints.

EqualityConstraintsException
    public EqualityConstraintsException(string message)
    The variables are determined by the equality constraints.

        message  The error message that explains the reason for the exception.

EqualityConstraintsException
    public EqualityConstraintsException(string s, System.Exception exception)
    The variables are determined by the equality constraints.

        s  The error message that explains the reason for the exception.
        exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

EqualityConstraintsException
    The variables are determined by the equality constraints.

        info  The object that holds the serialized object data.
        context  The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

FalseConvergenceException Class

False convergence, the iterates appear to be converging to a noncritical point.

public class Imsl.Math.FalseConvergenceException : IMSLEException implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException

public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message

virtual public string Message {get; }
Gets a message that describes the current exception.

Source
virtual public string Source {get; set; }
Gets or sets the name of the application or the object that causes the error.

StackTrace
virtual public string StackTrace {get; }
Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
public System.Reflection.MethodBase TargetSite {get; }
Gets the method that throws the current exception.

Constructors

FalseConvergenceException
public FalseConvergenceException()
False convergence, the iterates appear to be converging to a noncritical point.

FalseConvergenceException
public FalseConvergenceException(string message)
False convergence, the iterates appear to be converging to a noncritical point.

message The error message that explains the reason for the exception.

FalseConvergenceException
public FalseConvergenceException(string s, System.Exception exception)
False convergence, the iterates appear to be converging to a noncritical point.

s The error message that explains the reason for the exception.

exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

FalseConvergenceException
False convergence, the iterates appear to be converging to a noncritical point.

info The object that holds the serialized object data.

context The contextual information about the source or destination.
Methods

GetBaseException

```
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.
```

GetObjectData

```
virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the
```

ToString

```
override public string ToString()
Creates and returns a string representation of the current exception.
```

IllConditionedException Class

Problem is singular or ill-conditioned.

```
public class Imsl.Math.IllConditionedException : IMSLException implements
```

Properties

HResult

```
int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
```

HelpLink

```
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.
```

InnerException

```
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.
```

Message

```
virtual public string Message {get; }
```
Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

IllConditionedException
    public IllConditionedException()
    Problem is singular or ill-conditioned.

IllConditionedException
    public IllConditionedException(string message)
    Problem is singular or ill-conditioned.

        message  The error message that explains the reason for the exception.

IllConditionedException
    public IllConditionedException(string s, System.Exception exception)
    Problem is singular or ill-conditioned.

        s  The error message that explains the reason for the exception.

        exception  The exception that is the cause of the current exception. If the
                    innerException parameter is not a null reference, the current exception is raised in a
                    catch block that handles the inner exception.

IllConditionedException
    IllConditionedException(System.Runtime.Serialization.SerializationInfo
    Problem is singular or ill-conditioned.

        info  The object that holds the serialized object data.

        context  The contextual information about the source or destination.
Methods

GetBaseException

    virtual public System.Exception GetBaseException()
    When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

    virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
    When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

    override public string ToString()
    Creates and returns a string representation of the current exception.

InconsistentSystemException Class

Inconsistent system.

public class Imsl.Math.InconsistentSystemException : IMSLException implements

Properties

HResult

    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException

    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message

    virtual public string Message {get; }
Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

## Constructors

InconsistentSystemException

```csharp
public InconsistentSystemException()
```

Inconsistent system.

InconsistentSystemException

```csharp
public InconsistentSystemException(string message)
```

Inconsistent system.

- `message` The error message that explains the reason for the exception.

InconsistentSystemException

```csharp
public InconsistentSystemException(string s, System.Exception exception)
```

Inconsistent system.

- `s` The error message that explains the reason for the exception.
- `exception` The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

InconsistentSystemException

```csharp
```

Inconsistent system.

- `info` The object that holds the serialized object data.
- `context` The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.

GetObjectData

virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

LimitingAccuracyException Class

Limiting accuracy reached for a singular problem.

public class Imsl.Math.LimitingAccuracyException : IMSLException implements

Properties

HRESULT

int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException

public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message

virtual public string Message {get; }
Gets a message that describes the current exception.

**Source**

```csharp
virtual public string Source { get; set; }
```

Gets or sets the name of the application or the object that causes the error.

**StackTrace**

```csharp
virtual public string StackTrace { get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

**TargetSite**

```csharp
public System.Reflection.MethodBase TargetSite { get; }
```

Gets the method that throws the current exception.

### Constructors

**LimitingAccuracyException**

```csharp
public LimitingAccuracyException()
```

Limiting accuracy reached for a singular problem.

**LimitingAccuracyException**

```csharp
public LimitingAccuracyException(string message)
```

Limiting accuracy reached for a singular problem.

- **message** The error message that explains the reason for the exception.

**LimitingAccuracyException**

```csharp
public LimitingAccuracyException(string s, System.Exception exception)
```

Limiting accuracy reached for a singular problem.

- **s** The error message that explains the reason for the exception.
- **exception** The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**LimitingAccuracyException**

```csharp
```

Limiting accuracy reached for a singular problem.

- **info** The object that holds the serialized object data.
- **context** The contextual information about the source or destination.
**Methods**

GetBaseException

```csharp
virtual public System.Exception GetBaseException()
```
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

```csharp
```
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

```csharp
override public string ToString()
```
Creates and returns a string representation of the current exception.

---

**LinearlyDependentGradientsException Class**

Working set gradients are linearly dependent.

```csharp
public class Imsl.Math.LinearlyDependentGradientsException : IMSLEntity
```

**Properties**

HResult

```csharp
int HResult {get; set; }
```
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

```csharp
virtual public string HelpLink {get; set; }
```
Gets or sets a link to the help file associated with this exception.

InnerException

```csharp
public System.Exception InnerException {get; }
```
Gets the System.Exception instance that caused the current exception.

---

**Exceptions**

LinearlyDependentGradientsException Class • 801
Message
  
  virtual public string Message {get; }
  Gets a message that describes the current exception.

Source
  
  virtual public string Source {get; set; }
  Gets or sets the name of the application or the object that causes the error.

StackTrace
  
  virtual public string StackTrace {get; }
  Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
  
  public System.Reflection.MethodBase TargetSite {get; }
  Gets the method that throws the current exception.

Constructors

LinearlyDependentGradientsException
  
  public LinearlyDependentGradientsException()
  Working set gradients are linearly dependent.

LinearlyDependentGradientsException
  
  public LinearlyDependentGradientsException(string message)
  Working set gradients are linearly dependent.

  message The error message that explains the reason for the exception.

LinearlyDependentGradientsException
  
  public LinearlyDependentGradientsException(string s, System.Exception exception)
  Working set gradients are linearly dependent.

  s The error message that explains the reason for the exception.

  exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

LinearlyDependentGradientsException
  
  Working set gradients are linearly dependent.

  info The object that holds the serialized object data.

  context The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

MaxIterationsException Class

Maximum number of iterations exceeded.

public class Imsl.Math.MaxIterationsException : IMSLException implements

Properties

HResult

int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException

public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message

virtual public string Message {get; }
Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

## Constructors

MaxIterationsException

```csharp
public MaxIterationsException()
Maximum number of iterations exceeded.
```

MaxIterationsException

```csharp
public MaxIterationsException(string message)
Maximum number of iterations exceeded.
```

- **message** The error message that explains the reason for the exception.

MaxIterationsException

```csharp
public MaxIterationsException(string s, System.Exception exception)
Maximum number of iterations exceeded.
```

- **s** The error message that explains the reason for the exception.

- **exception** The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

MaxIterationsException

```csharp
Maximum number of iterations exceeded.
```

- **info** The object that holds the serialized object data.

- **context** The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

MaxNumberStepsAllowedException Class

Maximum number of steps allowed exceeded.

public class Imsl.Math.MaxNumberStepsAllowedException : IMSLException
implements

Properties

HResult

    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException

    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Exceptions
Message

    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source

    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace

    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

MaxNumberStepsAllowedException

    public MaxNumberStepsAllowedException()
    Maximum number of steps allowed exceeded.

MaxNumberStepsAllowedException

    public MaxNumberStepsAllowedException(int maxSteps)
    Maximum number of steps allowed exceeded.

        maxSteps    Maximum number of steps allowed.

MaxNumberStepsAllowedException

    public MaxNumberStepsAllowedException(string message)
    Maximum number of steps allowed exceeded.

        message    The error message that explains the reason for the exception.

MaxNumberStepsAllowedException

    public MaxNumberStepsAllowedException(string s, System.Exception exception)
    Maximum number of steps allowed exceeded.

        s    The error message that explains the reason for the exception.

        exception    The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

MaxNumberStepsAllowedException

    Maximum number of steps allowed exceeded.
info  The object that holds the serialized object data.
context  The contextual information about the source or destination.

Methods

GetBaseException
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData
virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString
override public string ToString()
Creates and returns a string representation of the current exception.

NoAcceptableStepsizeException Class

No acceptable stepsize in [SIGMA,SIGLA].

public class Imsl.Math.NoAcceptableStepsizeException : IMSLException
implements

Properties

HRESULT
int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.
InnerException

    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message

    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source

    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace

    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.

TargetSite

    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

NoAcceptableStepsizeException

    public NoAcceptableStepsizeException(double sigma, double sigla)
    No acceptable stepsize in [SIGMA,SIGLA].

    sigma    A double containing the first messages argument SIGMA.
    sigla    A double containing the second messages argument SIGLA.

NoAcceptableStepsizeException

    public NoAcceptableStepsizeException()
    No acceptable stepsize in [SIGMA,SIGLA].

NoAcceptableStepsizeException

    public NoAcceptableStepsizeException(string message)
    No acceptable stepsize in [SIGMA,SIGLA].

    message    The error message that explains the reason for the exception.

NoAcceptableStepsizeException

    public NoAcceptableStepsizeException(string s, System.Exception exception)
    No acceptable stepsize in [SIGMA,SIGLA].

    s    The error message that explains the reason for the exception.
    exception    The exception that is the cause of the current exception. If the
                  innerException parameter is not a null reference, the current exception is raised in a
                  catch block that handles the inner exception.
NoAcceptableStepsizeException


No acceptable stepsize in [SIGMA, SIGLA].

- info: The object that holds the serialized object data.
- context: The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

NotSPDEException Class

The matrix is not symmetric, positive definite.

public class Imsl.Math.NotSPDEException : IMSLException implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink
  
  virtual public string HelpLink {get; set; }
  
  Gets or sets a link to the help file associated with this exception.

InnerException
  
  public System.Exception InnerException {get; }
  
  Gets the System.Exception instance that caused the current exception.

Message
  
  virtual public string Message {get; }
  
  Gets a message that describes the current exception.

Source
  
  virtual public string Source {get; set; }
  
  Gets or sets the name of the application or the object that causes the error.

StackTrace
  
  virtual public string StackTrace {get; }
  
  Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
  
  public System.Reflection.MethodBase TargetSite {get; }
  
  Gets the method that throws the current exception.

Constructors

NotSPDException
  
  public NotSPDException()
  
  The matrix is not symmetric, positive definite.

NotSPDException
  
  public NotSPDException(string message)
  
  The matrix is not symmetric, positive definite.

    message  The error message that explains the reason for the exception.

NotSPDException
  
  public NotSPDException(string s, System.Exception exception)
  
  The matrix is not symmetric, positive definite.

    s  The error message that explains the reason for the exception.

    exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
NotSPDException

The matrix is not symmetric, positive definite.

- **info** The object that holds the serialized object data.
- **context** The contextual information about the source or destination.

**Methods**

GetBaseException

```csharp
virtual public System.Exception GetBaseException()
```
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

```csharp
```
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

```csharp
override public string ToString()
```
Creates and returns a string representation of the current exception.

---

**NumericDifficultyException Class**

Numerical difficulty occurred.

```csharp
public class Imsl.Math.NumericDifficultyException : IMSLException implements
```

**Properties**

**HResult**

```csharp
    int HResult {get; set; }
```
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink

```csharp
virtual public string HelpLink {get; set; }
```
Gets or sets a link to the help file associated with this exception.

InnerException

```csharp
public System.Exception InnerException {get; }
```
Gets the System.Exception instance that caused the current exception.

Message

```csharp
virtual public string Message {get; }
```
Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```
Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```
Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```
Gets the method that throws the current exception.

---

**Constructors**

NumericDifficultyException

```csharp
public NumericDifficultyException() 
```
Numerical difficulty occurred.

NumericDifficultyException

```csharp
public NumericDifficultyException(string message)
```
Numerical difficulty occurred.

**message** The error message that explains the reason for the exception.

NumericDifficultyException

```csharp
public NumericDifficultyException(string s, System.Exception exception)
```
Numerical difficulty occurred.

**s** The error message that explains the reason for the exception.

**exception** The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
NumericDifficultyException

 Numerical difficulty occurred.

 info  The object that holds the serialized object data.
 context  The contextual information about the source or destination.

Methods

GetBaseException

 virtual public System.Exception GetBaseException()
 When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

 virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
 When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

 override public string ToString()
 Creates and returns a string representation of the current exception.

ObjectiveEvaluationException Class

Objective evaluation returns an error with current point.

public class Imsl.Math.ObjectiveEvaluationException : IMSLException implements

Properties

HRESULT

 int HRESULT {get; set; }
 Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink

```csharp
virtual public string HelpLink {get; set; }
```

Gets or sets a link to the help file associated with this exception.

InnerException

```csharp
public System.Exception InnerException {get; }
```

Gets the System.Exception instance that caused the current exception.

Message

```csharp
virtual public string Message {get; }
```

Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

** Constructors **

ObjectiveEvaluationException

```csharp
public ObjectiveEvaluationException()
```

Objective evaluation returns an error with current point.

ObjectiveEvaluationException

```csharp
public ObjectiveEvaluationException(string message)
```

Objective evaluation returns an error with current point.

- ** message ** The error message that explains the reason for the exception.

ObjectiveEvaluationException

```csharp
public ObjectiveEvaluationException(string s, System.Exception exception)
```

Objective evaluation returns an error with current point.

- ** s ** The error message that explains the reason for the exception.
- ** exception ** The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
ObjectiveEvaluationException

Objective evaluation returns an error with current point.

- info: The object that holds the serialized object data.
- context: The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

PenaltyFunctionPointInfeasibleException Class

Penalty function point infeasible.

public class Imsl.Math.PenaltyFunctionPointInfeasibleException : IMSLEException implements

Properties

HResult

int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink

    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException

    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message

    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source

    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace

    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.

TargetSite

    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

PenaltyFunctionPointInfeasibleException

    public PenaltyFunctionPointInfeasibleException()
    Penalty function point infeasible.

PenaltyFunctionPointInfeasibleException

    public PenaltyFunctionPointInfeasibleException(string message)
    Penalty function point infeasible.

        message    The error message that explains the reason for the exception.

PenaltyFunctionPointInfeasibleException

    public PenaltyFunctionPointInfeasibleException(string s, System.Exception
    exception)
    Penalty function point infeasible.

        s    The error message that explains the reason for the exception.

        exception    The exception that is the cause of the current exception. If the
                   innerException parameter is not a null reference, the current exception is raised in a
                   catch block that handles the inner exception.
PenaltyFunctionPointInfeasibleException


Penalty function point infeasible.

info  The object that holds the serialized object data.
context  The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

ProblemInfeasibleException Class

The problem is not feasible. The constraints are inconsistent.

public class Imsl.Math.ProblemInfeasibleException : IMSLEException implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink
  virtual public string HelpLink {get; set; }
  Gets or sets a link to the help file associated with this exception.

InnerException
  public System.Exception InnerException {get; }
  Gets the System.Exception instance that caused the current exception.

Message
  virtual public string Message {get; }
  Gets a message that describes the current exception.

Source
  virtual public string Source {get; set; }
  Gets or sets the name of the application or the object that causes the error.

StackTrace
  virtual public string StackTrace {get; }
  Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
  public System.Reflection.MethodBase TargetSite {get; }
  Gets the method that throws the current exception.

Constructors

ProblemInfeasibleException
  public ProblemInfeasibleException()
  The problem is not feasible. The constraints are inconsistent.

ProblemInfeasibleException
  public ProblemInfeasibleException(string message)
  The problem is not feasible. The constraints are inconsistent.

message  The error message that explains the reason for the exception.

ProblemInfeasibleException
  public ProblemInfeasibleException(string s, System.Exception exception)
  The problem is not feasible. The constraints are inconsistent.

s  The error message that explains the reason for the exception.

exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
ProblemInfeasibleException

The problem is not feasible. The constraints are inconsistent.

info  The object that holds the serialized object data.
context  The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.

GetObjectData

virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

ProblemUnboundedException Class

The problem is unbounded.

public class Imsl.Math.ProblemUnboundedException : IMSLException implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

ProblemUnboundedException
    public ProblemUnboundedException()
    The problem is unbounded.

ProblemUnboundedException
    public ProblemUnboundedException(string message)
    The problem is unbounded.

        message  The error message that explains the reason for the exception.

ProblemUnboundedException
    public ProblemUnboundedException(string s, System.Exception exception)
    The problem is unbounded.

        s  The error message that explains the reason for the exception.

        exception  The exception that is the cause of the current exception. If the
                    innerException parameter is not a null reference, the current exception is raised in a
                    catch block that handles the inner exception.
ProblemUnboundedException

The problem is unbounded.

- **info** The object that holds the serialized object data.
- **context** The contextual information about the source or destination.

**Methods**

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

---

**QPInfeasibleException Class**

QP problem seemingly infeasible.

public class Imsl.Math.QPInfeasibleException : IMSLException implements

**Properties**

HResult

int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink

    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException

    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message

    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source

    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace

    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

QPInfeasibleException

    public QPInfeasibleException()
    QP problem seemingly infeasible.

QPInfeasibleException

    public QPInfeasibleException(string message)
    QP problem seemingly infeasible.

        message  The error message that explains the reason for the exception.

QPInfeasibleException

    public QPInfeasibleException(string s, System.Exception exception)
    QP problem seemingly infeasible.

        s  The error message that explains the reason for the exception.

        exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
QPInfeasibleException

QPInfeasibleException(System.Runtime.Serialization.SerializationInfo info,
QP problem seemingly infeasible.

info    The object that holds the serialized object data.
context The contextual information about the source or destination.

Methods

GetBaseException
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.

GetObjectData
virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString
override public string ToString()
Creates and returns a string representation of the current exception.

SingularException Class

Problem is singular.

public class Imsl.Math.SingularException : IMSLException implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

SingularException
    public SingularException()
    Problem is singular.

SingularException
    public SingularException(string message)
    Problem is singular.

        message  The error message that explains the reason for the exception.

SingularException
    public SingularException(string s, System.Exception exception)
    Problem is singular.

        s        The error message that explains the reason for the exception.

        exception  The exception that is the cause of the current exception. If the
                    innerException parameter is not a null reference, the current exception is raised in a
                    catch block that handles the inner exception.
SingularException

```csharp
SingularException(System.Runtime.Serialization.SerializationInfo info,
```

Problem is singular.

- **info**  The object that holds the serialized object data.
- **context**  The contextual information about the source or destination.

### Methods

**GetBaseException**

```csharp
virtual public System.Exception GetBaseException()
```

When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

**GetObjectData**

```csharp
virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
```

When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

**ToString**

```csharp
override public string ToString()
```

Creates and returns a string representation of the current exception.

---

**SingularMatrixException Class**

The matrix is singular.

```csharp
public class Imsl.Math.SingularMatrixException : IMSLErrException implements
```

### Properties

**HResult**

```csharp
int HResult {get; set; }
```

Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

---

**Exceptions**  SingularMatrixException Class • 825
HelpLink
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message
virtual public string Message {get; }
Gets a message that describes the current exception.

Source
virtual public string Source {get; set; }
Gets or sets the name of the application or the object that causes the error.

StackTrace
virtual public string StackTrace {get; }
Gets a string representation of the frames on the call stack at the time the current
exception was thrown.

TargetSite
public System.Reflection.MethodBase TargetSite {get; }
Gets the method that throws the current exception.

Constructors

SingularMatrixException
public SingularMatrixException()
The matrix is singular.

SingularMatrixException
public SingularMatrixException(string message)
The matrix is singular.

message The error message that explains the reason for the exception.

SingularMatrixException
public SingularMatrixException(string s, System.Exception exception)
The matrix is singular.

s The error message that explains the reason for the exception.

exception The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a
catch block that handles the inner exception.
SingularMatrixException


The matrix is singular.

**info**  The object that holds the serialized object data.
**context**  The contextual information about the source or destination.

### Methods

**GetBaseException**

*virtual public System.Exception GetBaseException()*

When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

**GetObjectData**


When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

**ToString**

*override public string ToString()*

Creates and returns a string representation of the current exception.

---

**TerminationCriteriaNotSatisfiedException Class**

Termination criteria are not satisfied.

*public class Imsl.Math.TerminationCriteriaNotSatisfiedException : IMSLException implements*

### Properties

**HR<sub>Result</sub>**

*int HResult {get; set; }*

Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }  
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }  
    Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }  
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }  
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get;} 
    Gets the method that throws the current exception.

Constructors

TerminationCriteriaNotSatisfiedException
    public TerminationCriteriaNotSatisfiedException(int numsm)
    Termination criteria are not satisfied.

        numsm    An int containing the criteria value.

TerminationCriteriaNotSatisfiedException
    public TerminationCriteriaNotSatisfiedException()
    Termination criteria are not satisfied.

TerminationCriteriaNotSatisfiedException
    public TerminationCriteriaNotSatisfiedException(string message)
    Termination criteria are not satisfied.

        message  The error message that explains the reason for the exception.

TerminationCriteriaNotSatisfiedException
    public TerminationCriteriaNotSatisfiedException(string s, System.Exception exception)
    Termination criteria are not satisfied.
The error message that explains the reason for the exception.

exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

TerminationCriteriaNotSatisfiedException
Termination criteria are not satisfied.

info  The object that holds the serialized object data.
context  The contextual information about the source or destination.

Methods

GetBaseException
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString
override public string ToString()
Creates and returns a string representation of the current exception.

ToleranceTooSmallException Class

Tolerance is too small.

public class Imsl.Math.ToleranceTooSmallException : IMSLException implements
Properties

HResult

```csharp
    int HResult {get; set; }
```

 Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

```csharp
    virtual public string HelpLink {get; set; }
```

 Gets or sets a link to the help file associated with this exception.

InnerException

```csharp
    public System.Exception InnerException {get; }
```

 Gets the System.Exception instance that caused the current exception.

Message

```csharp
    virtual public string Message {get; }
```

 Gets a message that describes the current exception.

Source

```csharp
    virtual public string Source {get; set; }
```

 Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
    virtual public string StackTrace {get; }
```

 Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
    public System.Reflection.MethodBase TargetSite {get; }
```

 Gets the method that throws the current exception.

Constructors

ToleranceTooSmallException

```csharp
    public ToleranceTooSmallException()
```

 Tolerance is too small.

ToleranceTooSmallException

```csharp
    public ToleranceTooSmallException(double tol)
```

 Tolerance is too small.

```csharp
    tol A double
    containing the tolerance value.
```
ToleranceTooSmallException

```csharp
public ToleranceTooSmallException(string message)
Tolerance is too small.

message The error message that explains the reason for the exception.
```

```csharp
public ToleranceTooSmallException(string s, System.Exception exception)
Tolerance is too small.

s The error message that explains the reason for the exception.

exception The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a
catch block that handles the inner exception.
```

```csharp
ToleranceTooSmallException
Tolerance is too small.

info The object that holds the serialized object data.

context The contextual information about the source or destination.
```

### Methods

**GetBaseException**

```csharp
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.
```

**GetObjectData**

```csharp
virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the
```

**ToString**

```csharp
override public string ToString()
Creates and returns a string representation of the current exception.
```
TooManyIterationsException Class

Maximum number of iterations exceeded.

```csharp
public class Imsl.Math.TooManyIterationsException : IMSLException implements Properties

HResult

    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException

    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message

    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source

    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace

    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.
```

Constructors

TooManyIterationsException

```csharp
public TooManyIterationsException(int maximumNumberOfIterations)
Maximum number of iterations exceeded.

    maximumNumberOfIterations Maximum number of iterations allowed.
```
TooManyIterationsException
    public TooManyIterationsException()
    Maximum number of iterations exceeded.

TooManyIterationsException
    public TooManyIterationsException(string message)
    Maximum number of iterations exceeded.
    message The error message that explains the reason for the exception.

TooManyIterationsException
    public TooManyIterationsException(string s, System.Exception exception)
    Maximum number of iterations exceeded.
    s The error message that explains the reason for the exception.
    exception The exception that is the cause of the current exception. If the
                innerException parameter is not a null reference, the current exception is raised in a
                catch block that handles the inner exception.

TooManyIterationsException
    TooManyIterationsException(System.Runtime.Serialization.SerializationInfo
    Maximum number of iterations exceeded.
    info The object that holds the serialized object data.
    context The contextual information about the source or destination.

Methods

GetBaseException
    virtual public System.Exception GetBaseException()
    When overridden in a derived class, returns the System.Exception that is the root cause
    of one or more subsequent exceptions.

GetObjectData
    virtual public void
    GetObjectData(System.Runtime.Serialization.SerializationInfo info,
    When overridden in a derived class, sets the

ToString
    override public string ToString()
    Creates and returns a string representation of the current exception.
UnboundedBelowException Class

Five consecutive steps of the maximum allowable stepsize have been taken, either the function is unbounded below, or has a finite asymptote in some direction or the maximum allowable step size is too small.

```csharp
public class Imsl.Math.UnboundedBelowException : IMSLException implements
```

**Properties**

HResult

```csharp
int HResult {get; set; }
```
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

```csharp
virtual public string HelpLink {get; set; }
```
Gets or sets a link to the help file associated with this exception.

InnerException

```csharp
public System.Exception InnerException {get; }
```
Gets the System.Exception instance that caused the current exception.

Message

```csharp
virtual public string Message {get; }
```
Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```
Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```
Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```
Gets the method that throws the current exception.

**Constructors**

```csharp
UnboundedBelowException
```
Five consecutive steps of the maximum allowable stepsize have been taken, either the function is unbounded below, or has a finite asymptote in some direction or the maximum allowable step size is too small.

UnboundedBelowException
public UnboundedBelowException(string message)
Five consecutive steps of the maximum allowable stepsize have been taken, either the function is unbounded below, or has a finite asymptote in some direction or the maximum allowable step size is too small.

message The error message that explains the reason for the exception.

UnboundedBelowException
public UnboundedBelowException(string s, System.Exception exception)
Five consecutive steps of the maximum allowable stepsize have been taken, either the function is unbounded below, or has a finite asymptote in some direction or the maximum allowable step size is too small.

s The error message that explains the reason for the exception.
exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

UnboundedBelowException
Five consecutive steps of the maximum allowable stepsize have been taken, either the function is unbounded below, or has a finite asymptote in some direction or the maximum allowable step size is too small.

info The object that holds the serialized object data.
context The contextual information about the source or destination.

Methods

GetBaseException
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData
virtual public void
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.
ToString
override public string ToString()
Creates and returns a string representation of the current exception.

VarBoundsInconsistentException Class

The equality constraints and the bounds on the variables are found to be inconsistent.

public class Imsl.Math.VarBoundsInconsistentException : IMSLEException
implements

Properties

HResult
int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message
virtual public string Message {get; }
Gets a message that describes the current exception.

Source
virtual public string Source {get; set; }
Gets or sets the name of the application or the object that causes the error.

StackTrace
virtual public string StackTrace {get; }
Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
public System.Reflection.MethodBase TargetSite {get; }
Gets the method that throws the current exception.
Constructors

VarBoundsInconsistentException
public VarBoundsInconsistentException()
The equality constraints and the bounds on the variables are found to be inconsistent.

VarBoundsInconsistentException
public VarBoundsInconsistentException(string message)
The equality constraints and the bounds on the variables are found to be inconsistent.
message The error message that explains the reason for the exception.

VarBoundsInconsistentException
public VarBoundsInconsistentException(string s, System.Exception exception)
The equality constraints and the bounds on the variables are found to be inconsistent.
s The error message that explains the reason for the exception.
exception The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a
catch block that handles the inner exception.

VarBoundsInconsistentException
The equality constraints and the bounds on the variables are found to be inconsistent.
info The object that holds the serialized object data.
context The contextual information about the source or destination.

Methods

GetBaseException
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.

GetObjectData
virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString
override public string ToString()
Creates and returns a string representation of the current exception.
WorkingSetSingularException Class

Working set is singular in dual extended QP.

```csharp
public class Imsl.Math.WorkingSetSingularException : IMSLException implements

Properties

HResult

```csharp
    int HResult {get; set; }
```

Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

```csharp
    virtual public string HelpLink {get; set; }
```

Gets or sets a link to the help file associated with this exception.

InnerException

```csharp
    public System.Exception InnerException {get; }
```

Gets the System.Exception instance that caused the current exception.

Message

```csharp
    virtual public string Message {get; }
```

Gets a message that describes the current exception.

Source

```csharp
    virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
    virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
    public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

Constructors

WorkingSetSingularException

```csharp
    public WorkingSetSingularException()
    
    Working set is singular in dual extended QP.
```
WorkingSetSingularException
   public WorkingSetSingularException(string message)
   Working set is singular in dual extended QP.

   message The error message that explains the reason for the exception.

WorkingSetSingularException
   public WorkingSetSingularException(string s, System.Exception exception)
   Working set is singular in dual extended QP.

   s The error message that explains the reason for the exception.
   exception The exception that is the cause of the current exception. If the
   innerException parameter is not a null reference, the current exception is raised in a
   catch block that handles the inner exception.

WorkingSetSingularException
   WorkingSetSingularException(System.Runtime.Serialization.SerializationInfo
   Working set is singular in dual extended QP.

   info The object that holds the serialized object data.
   context The contextual information about the source or destination.

Methods

GetBaseException
   virtual public System.Exception GetBaseException()
   When overridden in a derived class, returns the System.Exception that is the root cause
   of one or more subsequent exceptions.

GetObjectData
   virtual public void
   GetObjectData(System.Runtime.Serialization.SerializationInfo info,
   When overridden in a derived class, sets the

ToString
   override public string ToString()
   Creates and returns a string representation of the current exception.
AllDeletedException Class

There are no observations.

```csharp
public class Imsl.Stat.AllDeletedException : IMSLException implements Properties

HResult
    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

 InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

 Message
    virtual public string Message {get; }
    Gets a message that describes the current exception.

 Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

 StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

 TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.
```

Constructors

AllDeletedException
    public AllDeletedException()
    There are no observations.
AllDeletedException
public AllDeletedException(string message)
There are no observations.

message   The error message that explains the reason for the exception.

AllDeletedException
public AllDeletedException(string s, System.Exception exception)
There are no observations.

s       The error message that explains the reason for the exception.
exception The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a

catch block that handles the inner exception.

AllDeletedException
AllDeletedException(System.Runtime.Serialization.SerializationInfo info,
There are no observations.

info   The object that holds the serialized object data.
context The contextual information about the source or destination.

Methods

GetBaseException
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.

GetObjectData
virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString
override public string ToString()
Creates and returns a string representation of the current exception.
AllMissingException Class

There are no observations.

public class Imsl.Stat.AllMissingException : IMSLException implements

Properties

HRESULT
int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message
virtual public string Message {get; }
Gets a message that describes the current exception.

Source
virtual public string Source {get; set; }
Gets or sets the name of the application or the object that causes the error.

StackTrace
virtual public string StackTrace {get; }
Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
public System.Reflection.MethodBase TargetSite {get; }
Gets the method that throws the current exception.

Constructors

AllMissingException
public AllMissingException()
There are no observations.
AllMissingException
    public AllMissingException(string message)
    There are no observations.

    message  The error message that explains the reason for the exception.

AllMissingException
    public AllMissingException(string s, System.Exception exception)
    There are no observations.

    s   The error message that explains the reason for the exception.
    exception  The exception that is the cause of the current exception. If the
               innerException parameter is not a null reference, the current exception is raised in
               a catch block that handles the inner exception.

AllMissingException
    AllMissingException(System.Runtime.Serialization.SerializationInfo info,
    There are no observations.

    info  The object that holds the serialized object data.
    context  The contextual information about the source or destination.

Methods

GetBaseException
    virtual public System.Exception GetBaseException()
    When overridden in a derived class, returns the System.Exception that is the root cause
    of one or more subsequent exceptions.

GetObjectData
    virtual public void
    GetObjectData(System.Runtime.Serialization.SerializationInfo info,
    When overridden in a derived class, sets the

ToString
    override public string ToString()
    Creates and returns a string representation of the current exception.
BadVarianceException Class

The input variance is not in the allowed range.

public class Imsl.Stat.BadVarianceException : IMSLException implements

Properties

HRESULT

    int HRESULT {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException

    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message

    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source

    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace

    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

BadVarianceException

    public BadVarianceException(int i, double cov, double uniq)
    The input variance is not in the allowed range.

        i     A int specifying the index of variable uniq, causing the error.
cov A double specifying the value of cov[i,i].
uniq A double specifying the input variance.

BadVarianceException
   public BadVarianceException()
   Maximum number of iterations exceeded.

BadVarianceException
   public BadVarianceException(string message)
   Maximum number of iterations exceeded.
   
   message The error message that explains the reason for the exception.

BadVarianceException
   public BadVarianceException(string s, System.Exception exception)
   Maximum number of iterations exceeded.
   
   s The error message that explains the reason for the exception.
   exception The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a
catch block that handles the inner exception.

BadVarianceException
   BadVarianceException(System.Runtime.Serialization.SerializationInfo info,
   Maximum number of iterations exceeded.

   info The object that holds the serialized object data.
   context The contextual information about the source or destination.

Methods

GetBaseException
   virtual public System.Exception GetBaseException()
   When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.

GetObjectData
   virtual public void
   GetObjectData(System.Runtime.Serialization.SerializationInfo info,
   When overridden in a derived class, sets the
ToString

```csharp
override public string ToString()
    Creates and returns a string representation of the current exception.
```

---

**ClassificationVariableException Class**

The ClassificationVariable vector has not been initialized.

```csharp
public class Imsl.Stat.ClassificationVariableException : IMSLException
    implements
```

**Properties**

**HRresult**

```csharp
    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
```

**HelpLink**

```csharp
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.
```

**InnerException**

```csharp
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.
```

**Message**

```csharp
    virtual public string Message {get; }
    Gets a message that describes the current exception.
```

**Source**

```csharp
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.
```

**StackTrace**

```csharp
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.
```

**TargetSite**

```csharp
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.
```
Constructors

ClassificationVariableException

public ClassificationVariableException()
The ClassificationVariable vector has not been initialized.

ClassificationVariableException

public ClassificationVariableException(string message)
The ClassificationVariable vector has not been initialized.

message The error message that explains the reason for the exception.

ClassificationVariableException

public ClassificationVariableException(string s, System.Exception exception)
The ClassificationVariable vector has not been initialized.

s The error message that explains the reason for the exception.

exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

ClassificationVariableException

The ClassificationVariable vector has not been initialized.

info The object that holds the serialized object data.

context The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.
ToString

    override public string ToString()
    Creates and returns a string representation of the current exception.

ClassificationVariableLimitException Class

The Classification Variable limit set by the user through `setUpperBound` has been exceeded.

    public class Imsl.Stat.ClassificationVariableLimitException : IMSLException
    implements

Properties

    HResult
        int HResult {get; set; }
        Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

    HelpLink
        virtual public string HelpLink {get; set; }
        Gets or sets a link to the help file associated with this exception.

    InnerException
        public System.Exception InnerException {get; }
        Gets the System.Exception instance that caused the current exception.

    Message
        virtual public string Message {get; }
        Gets a message that describes the current exception.

    Source
        virtual public string Source {get; set; }
        Gets or sets the name of the application or the object that causes the error.

    StackTrace
        virtual public string StackTrace {get; }
        Gets a string representation of the frames on the call stack at the time the current
        exception was thrown.

    TargetSite
        public System.Reflection.MethodBase TargetSite {get; }
        Gets the method that throws the current exception.
### Constructors

**ClassificationVariableLimitException**

- **public ClassificationVariableLimitException(int maxcl)**
  The Classification Variable limit set by the user through `setUpperBound` has been exceeded.

  **maxcl**  An `int` which specifies the upper bound.

- **public ClassificationVariableLimitException()**
  The Classification Variable limit set by the user through `setUpperBound` has been exceeded.

- **public ClassificationVariableLimitException(string message)**
  The Classification Variable limit set by the user through `setUpperBound` has been exceeded.

  **message**  The error message that explains the reason for the exception.

- **public ClassificationVariableLimitException(string s, System.Exception exception)**
  The Classification Variable limit set by the user through `setUpperBound` has been exceeded.

  **s**  The error message that explains the reason for the exception.

  **exception**  The exception that is the cause of the current exception. If the `innerException` parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

  The Classification Variable limit set by the user through `setUpperBound` has been exceeded.

  **info**  The object that holds the serialized object data.

  **context**  The contextual information about the source or destination.

### Methods

**GetBaseException**

- **virtual public System.Exception GetBaseException()**
  When overridden in a derived class, returns the `System.Exception` that is the root cause of one or more subsequent exceptions.
GetObjectData

```csharp
virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the
```

ToString

```csharp
override public string ToString()
Creates and returns a string representation of the current exception.
```

---

**ClassificationVariableValueException Class**

The number of distinct values for each Classification Variable must be greater than 1.

```csharp
public class Imsl.Stat.ClassificationVariableValueException : IMSLException
implements
```

**Properties**

HResult

```csharp
int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
```

HelpLink

```csharp
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.
```

InnerException

```csharp
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.
```

Message

```csharp
virtual public string Message {get; }
Gets a message that describes the current exception.
```

Source

```csharp
virtual public string Source {get; set; }
Gets or sets the name of the application or the object that causes the error.
```
StackTrace

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

**Constructors**

ClassificationVariableValueException

```csharp
public ClassificationVariableValueException(int index, int val)
```

The number of distinct values for each Classification Variable must be greater than 1.

- **index**  An int which specifies the index of a classification variable.
- **val** An int which specifies the number of distinct values that can be taken by this classification variable.

ClassificationVariableValueException

```csharp
public ClassificationVariableValueException()
```

The number of distinct values for each Classification Variable must be greater than 1.

ClassificationVariableValueException

```csharp
public ClassificationVariableValueException(string message)
```

The number of distinct values for each Classification Variable must be greater than 1.

- **message** The error message that explains the reason for the exception.

ClassificationVariableValueException

```csharp
public ClassificationVariableValueException(string s, System.Exception exception)
```

The number of distinct values for each Classification Variable must be greater than 1.

- **s** The error message that explains the reason for the exception.
- **exception** The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

ClassificationVariableValueException

```csharp
```

The number of distinct values for each Classification Variable must be greater than 1.

- **info** The object that holds the serialized object data.
- **context** The contextual information about the source or destination.
Methods

GetBaseException

```csharp
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.
```

GetObjectData

```csharp
virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the
```

ToString

```csharp
override public string ToString()
Creates and returns a string representation of the current exception.
```

ClusterNoPointsException Class

There is a cluster with no points.

```csharp
public class Imsl.Stat.ClusterNoPointsException : IMSLException implements
```

Properties

HResult

```csharp
int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
```

HelpLink

```csharp
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.
```

InnerException

```csharp
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.
```

Message

```csharp
virtual public string Message {get; }
```
Gets a message that describes the current exception.

Source
virtual public string Source {get; set; }
Gets or sets the name of the application or the object that causes the error.

StackTrace
virtual public string StackTrace {get; }
Gets a string representation of the frames on the call stack at the time the current
exception was thrown.

TargetSite
public System.Reflection.MethodBase TargetSite {get; }
Gets the method that throws the current exception.

Constructors

ClusterNoPointsException
public ClusterNoPointsException()
There is a cluster with no points.

ClusterNoPointsException
public ClusterNoPointsException(int clusterNumber)
There is a cluster with no points.

  clusterNumber  Number of the cluster with no points.

ClusterNoPointsException
public ClusterNoPointsException(string message)
There is a cluster with no points.

  message  The error message that explains the reason for the exception.

ClusterNoPointsException
public ClusterNoPointsException(string s, System.Exception exception)
There is a cluster with no points.

  s  The error message that explains the reason for the exception.

  exception  The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a
catch block that handles the inner exception.

ClusterNoPointsException
ClusterNoPointsException(System.Runtime.Serialization.SerializationInfo
There is a cluster with no points.

  info  The object that holds the serialized object data.

  context  The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.

GetObjectData

virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

ConstrInconsistentException Class

The equality constraints are inconsistent.

public class Imsl.Stat.ConstrInconsistentException : IMSLException implements

Properties

HResult

int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException

public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message

virtual public string Message {get; }
Gets a message that describes the current exception.

**Source**

```
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

**StackTrace**

```
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

**TargetSite**

```
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

---

### Constructors

**ConstrInconsistentException**

```
public ConstrInconsistentException()
```

The equality constraints are inconsistent.

**ConstrInconsistentException**

```
public ConstrInconsistentException(string message)
```

The equality constraints are inconsistent.

- **message**  The error message that explains the reason for the exception.

**ConstrInconsistentException**

```
public ConstrInconsistentException(string s, System.Exception exception)
```

The equality constraints are inconsistent.

- **s**  The error message that explains the reason for the exception.
- **exception**  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**ConstrInconsistentException**

```
```

The equality constraints are inconsistent.

- **info**  The object that holds the serialized object data.
- **context**  The contextual information about the source or destination.
Methods

GetBaseException
    virtual public System.Exception GetBaseException()
    When overridden in a derived class, returns the System.Exception that is the root cause
    of one or more subsequent exceptions.

GetObjectData
    virtual public void
    GetObjectData(System.Runtime.Serialization.SerializationInfo info,
    When overridden in a derived class, sets the

ToString
    override public string ToString()
    Creates and returns a string representation of the current exception.

CovarianceSingularException Class

The variance-Covariance matrix is singular.

public class Imsl.Stat.CovarianceSingularException : IMSLException implements

Properties

HResult
    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }
Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

CovarianceSingularException
    public CovarianceSingularException()
    The variance-Covariance matrix is singular.

CovarianceSingularException
    public CovarianceSingularException(int l)
    The variance-Covariance matrix is singular.

    l  A int which specifies the population number.

CovarianceSingularException
    public CovarianceSingularException(string message)
    The variance-Covariance matrix is singular.

    message  The error message that explains the reason for the exception.

CovarianceSingularException
    public CovarianceSingularException(string s, System.Exception exception)
    The variance-Covariance matrix is singular.

    s  The error message that explains the reason for the exception.

    exception  The exception that is the cause of the current exception. If the
               innerException parameter is not a null reference, the current exception is raised in a
               catch block that handles the inner exception.

CovarianceSingularException
    CovarianceSingularException(System.Runtime.Serialization.SerializationInfo
    The variance-Covariance matrix is singular.

    info  The object that holds the serialized object data.

    context  The contextual information about the source or destination.
Methods

GetBaseException
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData
virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString
override public string ToString()
Creates and returns a string representation of the current exception.

CyclingIsOccurringException Class

Cycling is occurring.

public class Imsl.Stat.CyclingIsOccurringException : IMSLException implements

Properties

HRESULT
int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message
virtual public string Message {get; }
Gets a message that describes the current exception.

**Source**

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

**StackTrace**

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

**TargetSite**

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

### Constructors

**CyclingIsOccurringException**

```csharp
public CyclingIsOccurringException(int nStep)
```

Cycling is occurring.

- **nStep**  An `int` which specifies the number of steps taken.

**CyclingIsOccurringException**

```csharp
public CyclingIsOccurringException()
```

Cycling is occurring.

**CyclingIsOccurringException**

```csharp
public CyclingIsOccurringException(string message)
```

Cycling is occurring.

- **message**  The error message that explains the reason for the exception.

**CyclingIsOccurringException**

```csharp
public CyclingIsOccurringException(string s, System.Exception exception)
```

Cycling is occurring.

- **s**  The error message that explains the reason for the exception.
- **exception**  The exception that is the cause of the current exception. If the `innerException` parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**CyclingIsOccurringException**

```csharp
```

Cycling is occurring.

- **info**  The object that holds the serialized object data.
- **context**  The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.

GetObjectData

virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

DeleteObservationsException Class

The number of observations to be deleted (set by setObservationMax) has grown too large.

public class Imsl.Stat.DeleteObservationsException : IMSLException implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException

public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message

virtual public string Message {get; }
Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

## Constructors

DeleteObservationsException

```csharp
public DeleteObservationsException(int nmax)
```

The number of observations to be deleted (set with ObservationMax) has grown too large.

- **nmax** An int which specifies the maximum number of observations that can be handled in the linear programming.

DeleteObservationsException

```csharp
public DeleteObservationsException()
```

The number of observations to be deleted (set by setObservationMax) has grown too large.

DeleteObservationsException

```csharp
public DeleteObservationsException(string message)
```

The number of observations to be deleted (set by setObservationMax) has grown too large.

- **message** The error message that explains the reason for the exception.

DeleteObservationsException

```csharp
public DeleteObservationsException(string s, System.Exception exception)
```

The number of observations to be deleted (set by setObservationMax) has grown too large.

- s The error message that explains the reason for the exception.
- exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
DeleteObservationsException

The number of observations to be deleted (set by setObservationMax) has grown too large.

  info  The object that holds the serialized object data.
  context  The contextual information about the source or destination.

Methods

GetBaseException
  virtual public System.Exception GetBaseException()
  When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData
  virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
  When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString
  override public string ToString()
  Creates and returns a string representation of the current exception.

---

DidNotConvergeException Class

The iteration did not converge.

public class Imsl.Stat.DidNotConvergeException : IMSLException implements

Properties

HResult
  int HResult {get; set; }
  Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink
  virtual public string HelpLink {get; set; }
  Gets or sets a link to the help file associated with this exception.

InnerException
  public System.Exception InnerException {get; }
  Gets the System.Exception instance that caused the current exception.

Message
  virtual public string Message {get; }
  Gets a message that describes the current exception.

Source
  virtual public string Source {get; set; }
  Gets or sets the name of the application or the object that causes the error.

StackTrace
  virtual public string StackTrace {get; }
  Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
  public System.Reflection.MethodBase TargetSite {get; }
  Gets the method that throws the current exception.

Constructors

DidNotConvergeException
  public DidNotConvergeException()
  The iteration did not converge.

DidNotConvergeException
  public DidNotConvergeException(string message)
  The iteration did not converge.

      message  The error message that explains the reason for the exception.

DidNotConvergeException
  public DidNotConvergeException(string s, System.Exception exception)
  The iteration did not converge.

      s  The error message that explains the reason for the exception.

      exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
DidNotConvergeException


The iteration did not converge.

info The object that holds the serialized object data.
context The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()

When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData


When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()

Creates and returns a string representation of the current exception.

DiffObsDeletedException Class

Different observations are being deleted from return matrix than were originally entered.

public class Imsl.Stat.DiffObsDeletedException : IMSLException implements

Properties

HResult

int HResult {get; set; }

Obtains or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

DiffObsDeletedException
    public DiffObsDeletedException()
    Different observations are being deleted from return matrix than were originally entered.

DiffObsDeletedException
    public DiffObsDeletedException(int i)
    Different observations are being deleted from return matrix than were originally entered.

        i  An int which specifies the index of Variance-Covariance matrix.

DiffObsDeletedException
    public DiffObsDeletedException(string message)
    Different observations are being deleted from return matrix than were originally entered.

        message  The error message that explains the reason for the exception.

DiffObsDeletedException
    public DiffObsDeletedException(string s, System.Exception exception)
    Different observations are being deleted from return matrix than were originally entered.

        s  The error message that explains the reason for the exception.
exception  The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a
catch block that handles the inner exception.

DiffObsDeletedException
DiffObsDeletedException(System.Runtime.Serialization.SerializationInfo
Different observations are being deleted from return matrix than were originally entered.

info  The object that holds the serialized object data.
context  The contextual information about the source or destination.

Methods

GetBaseException
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.

GetObjectData
virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString
override public string ToString()
Creates and returns a string representation of the current exception.

EigenvalueException Class

An error occurred in calculating the eigenvalues of the adjusted (inverse) covariance matrix.
Check the input covariance matrix.

public class Imsl.Stat.EigenvalueException : IMSLException implements

866 • EigenvalueException Class  IMSL C# Numerical Library
Properties

HResult

```csharp
int HResult {get; set; }
```

Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

```csharp
virtual public string HelpLink {get; set; }
```

Gets or sets a link to the help file associated with this exception.

InnerException

```csharp
public System.Exception InnerException {get; }
```

Gets the System.Exception instance that caused the current exception.

Message

```csharp
virtual public string Message {get; }
```

Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

Constructors

EigenvalueException

```csharp
public EigenvalueException()
```

Eigenvalue error.

EigenvalueException

```csharp
public EigenvalueException(string message)
```

Eigenvalue error.

message  The error message that explains the reason for the exception.

EigenvalueException

```csharp
public EigenvalueException(string s, System.Exception exception)
```

Eigenvalue error.

s  The error message that explains the reason for the exception.
exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

EigenvalueException

Eigenvalue error.

info  The object that holds the serialized object data.
context  The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

EmptyGroupException Class

There are no observations in a group. Cannot compute statistics.

public class Imsl.Stat.EmptyGroupException : IMSLException implements
Properties

HRESULT
int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message
virtual public string Message {get; }
Gets a message that describes the current exception.

Source
virtual public string Source {get; set; }
Gets or sets the name of the application or the object that causes the error.

StackTrace
virtual public string StackTrace {get; }
Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
public System.Reflection.MethodBase TargetSite {get; }
Gets the method that throws the current exception.

Constructors

EmptyGroupException
public EmptyGroupException(int group)
There are no observations in a group. Cannot compute statistics.

    group A int which specifies the index of empty group.

EmptyGroupException
public EmptyGroupException()
There are no observations in a group. Cannot compute statistics.

EmptyGroupException
public EmptyGroupException(string message)
There are no observations in a group. Cannot compute statistics.

    message The error message that explains the reason for the exception.

EmptyGroupException
public EmptyGroupException(string s, System.Exception exception)
There are no observations in a group. Cannot compute statistics.
EmptyGroupException

There are no observations in a group. Cannot compute statistics.

info The object that holds the serialized object data.
context The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

EqConstrInconsistentException Class

The equality constraints and the bounds on the variables are found to be inconsistent.

public class Imsl.Stat.EqConstrInconsistentException : IMSLException
implements
Properties

HRESULT
  int HRESULT {get; set; }
  Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
  virtual public string HelpLink {get; set; }
  Gets or sets a link to the help file associated with this exception.

InnerException
  public System.Exception InnerException {get; }
  Gets the System.Exception instance that caused the current exception.

Message
  virtual public string Message {get; }
  Gets a message that describes the current exception.

Source
  virtual public string Source {get; set; }
  Gets or sets the name of the application or the object that causes the error.

StackTrace
  virtual public string StackTrace {get; }
  Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
  public System.Reflection.MethodBase TargetSite {get; }
  Gets the method that throws the current exception.

Constructors

EqConstrInconsistentException
  public EqConstrInconsistentException()
  The equality constraints and the bounds on the variables are found to be inconsistent.

EqConstrInconsistentException
  public EqConstrInconsistentException(string message)
  The equality constraints and the bounds on the variables are found to be inconsistent.

    message  The error message that explains the reason for the exception.

EqConstrInconsistentException
  public EqConstrInconsistentException(string s, System.Exception exception)
  The equality constraints and the bounds on the variables are found to be inconsistent.

    s  The error message that explains the reason for the exception.
**exception**  The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a
  catch block that handles the inner exception.

EqConstrInconsistentException
The equality constraints and the bounds on the variables are found to be inconsistent.

  info  The object that holds the serialized object data.
  context  The contextual information about the source or destination.

**Methods**

GetBaseException
  virtual public System.Exception GetBaseException()
  When overridden in a derived class, returns the System.Exception that is the root cause
  of one or more subsequent exceptions.

GetObjectData
  virtual public void
  GetObjectData(System.Runtime.Serialization.SerializationInfo info,
  When overridden in a derived class, sets the

ToString
  override public string ToString()
  Creates and returns a string representation of the current exception.

**IllConditionedException Class**

The problem is ill-conditioned.

public class Imsl.Stat.IllConditionedException : IMSLException implements

**Properties**

HResult
int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

IllConditionedException
    public IllConditionedException()
    The problem is ill-conditioned.

IllConditionedException
    public IllConditionedException(string message)
    The problem is ill-conditioned.

        message The error message that explains the reason for the exception.

IllConditionedException
    public IllConditionedException(string s, System.Exception exception)
    The problem is ill-conditioned.

        s The error message that explains the reason for the exception.
        exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
IllConditionedException

The problem is ill-conditioned.

- **info**  The object that holds the serialized object data.
- **context**  The contextual information about the source or destination.

**Methods**

**GetBaseException**

`virtual public System.Exception GetBaseException()`  
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

**GetObjectData**

When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

**ToString**

`override public string ToString()`  
Creates and returns a string representation of the current exception.

---

**IncreaseErrRelException Class**

The bound for the relative error is too small.

```csharp
public class Imsl.Stat.IncreaseErrRelException : IMSLException implements
```

**Properties**

**HResult**

```csharp
int HResult {get; set; }
```

Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

IncreaseErrRelException
    public IncreaseErrRelException(double relativeError)
    The bound for the relative error is too small.

    relativeError A double which specifies the bound for relative error.

IncreaseErrRelException
    public IncreaseErrRelException()
    The bound for the relative error is too small.

IncreaseErrRelException
    public IncreaseErrRelException(string message)
    The bound for the relative error is too small.

    message The error message that explains the reason for the exception.

IncreaseErrRelException
    public IncreaseErrRelException(string s, System.Exception exception)
    The bound for the relative error is too small.

    s The error message that explains the reason for the exception.
exception The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a
catch block that handles the inner exception.

IncreaseErrRelException

IncreaseErrRelException(System.Runtime.Serialization.SerializationInfo info,
The bound for the relative error is too small.

info The object that holds the serialized object data.

context The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.

GetObjectData

virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

MatrixSingularException Class

The input matrix is singular.

public class Imsl.Stat.MatrixSingularException : IMSLException implements

Properties

HRESULT
int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
    virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }
Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
Gets the method that throws the current exception.

 Constructors

MatrixSingularException
    public MatrixSingularException()
The input matrix is singular.

MatrixSingularException
    public MatrixSingularException(string message)
The input matrix is singular.

        message    The error message that explains the reason for the exception.

MatrixSingularException
    public MatrixSingularException(string s, System.Exception exception)
The input matrix is singular.

        s      The error message that explains the reason for the exception.

        exception    The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
MatrixSingularException

The input matrix is singular.

info  The object that holds the serialized object data.
context  The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.

GetObjectData

virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

MoreObsDelThanEnteredException Class

More observations are being deleted from the output covariance matrix than were originally
entered (the corresponding row, column of the incidence matrix is less than zero).

public class Imsl.Stat.MoreObsDelThanEnteredException : IMSLEException
implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink
  virtual public string HelpLink {get; set; }
  Gets or sets a link to the help file associated with this exception.

InnerException
  public System.Exception InnerException {get; }
  Gets the System.Exception instance that caused the current exception.

Message
  virtual public string Message {get; }
  Gets a message that describes the current exception.

Source
  virtual public string Source {get; set; }
  Gets or sets the name of the application or the object that causes the error.

StackTrace
  virtual public string StackTrace {get; }
  Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
  public System.Reflection.MethodBase TargetSite {get; }
  Gets the method that throws the current exception.

Constructors

MoreObsDelThanEnteredException
  public MoreObsDelThanEnteredException(int j, int k)
  More observations are being deleted from the output covariance matrix than were originally entered (the corresponding row, column of the incidence matrix is less than zero).

  j  A int which specifies the row index of Variance-Covariance matrix.

  k  A int which specifies the column index of Variance-Covariance matrix.

MoreObsDelThanEnteredException
  public MoreObsDelThanEnteredException()
  More observations are being deleted from the output covariance matrix than were originally entered (the corresponding row, column of the incidence matrix is less than zero).

MoreObsDelThanEnteredException
  public MoreObsDelThanEnteredException(string message)
  More observations are being deleted from the output covariance matrix than were originally entered (the corresponding row, column of the incidence matrix is less than zero).
**message**  The error message that explains the reason for the exception.

MoreObsDelThanEnteredException

```csharp
public MoreObsDelThanEnteredException(string s, System.Exception exception)
More observations are being deleted from the output covariance matrix than were
originally entered (the corresponding row, column of the incidence matrix is less than
zero).
```

- **s**  The error message that explains the reason for the exception.
- **exception**  The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a
catch block that handles the inner exception.

MoreObsDelThanEnteredException

```csharp
MoreObsDelThanEnteredException(System.Runtime.Serialization.SerializationInfo
More observations are being deleted from the output covariance matrix than were
originally entered (the corresponding row, column of the incidence matrix is less than
zero).
```

- **info**  The object that holds the serialized object data.
- **context**  The contextual information about the source or destination.

**Methods**

GetBaseException

```csharp
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.
```

GetObjectData

```csharp
virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the
```

ToString

```csharp
override public string ToString()
Creates and returns a string representation of the current exception.
```
NegativeFreqException Class

A negative frequency was encountered.

```csharp
public class Imsl.Stat.NegativeFreqException : IMSLException implements
```

**Properties**

**HRESULT**

```csharp
int HResult {get; set; }
```

Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

**HelpLink**

```csharp
virtual public string HelpLink {get; set; }
```

Gets or sets a link to the help file associated with this exception.

**InnerException**

```csharp
public System.Exception InnerException {get; }
```

Gets the System.Exception instance that caused the current exception.

**Message**

```csharp
virtual public string Message {get; }
```

Gets a message that describes the current exception.

**Source**

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

**StackTrace**

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

**TargetSite**

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

**Constructors**

**NegativeFreqException**

```csharp
public NegativeFreqException(int rowIndex, int invocation, double val)
```

A negative frequency was encountered.

**Exceptions**

- NegativeFreqException Class • 881
**rowIndex**  An int which specifies the row index of X for which the frequency is negative.

**invocation**  An int which specifies the invocation number at which the error occurred.
   A 3 would indicate that the error occurred on the third invocation.

**val**  A double which represents the value of the frequency encountered.

NegativeFreqException
    public NegativeFreqException()
    A negative frequency was encountered.

NegativeFreqException
    public NegativeFreqException(string message)
    A negative frequency was encountered.

    **message**  The error message that explains the reason for the exception.

NegativeFreqException
    public NegativeFreqException(string s, System.Exception exception)
    A negative frequency was encountered.

    **s**  The error message that explains the reason for the exception.

    **exception**  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

NegativeFreqException
    A negative frequency was encountered.

    **info**  The object that holds the serialized object data.

    **context**  The contextual information about the source or destination.

**Methods**

GetBaseException
    virtual public System.Exception GetBaseException()
    When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData
    virtual public void
    When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.
ToString

    override public string ToString()
    Creates and returns a string representation of the current exception.

---

**NegativeWeightException Class**

A negative weight was encountered.

```csharp
public class Imsl.Stat.NegativeWeightException : IMSLException implements
```

**Properties**

**HResult**

    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

**HelpLink**

    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

**InnerException**

    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

**Message**

    virtual public string Message {get; }
    Gets a message that describes the current exception.

**Source**

    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

**StackTrace**

    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.

**TargetSite**

    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.
Constructors

public NegativeWeightException(int rowIndex, int invocation, double val)
A negative weight was encountered.

rowIndex An int which specifies the row index of X for which the weight is negative.

invocation An int which specifies the invocation number at which the error occurred.
A 3 would indicate that the error occurred on the third invocation.

val An double which represents the value of the weight encountered.

public NegativeWeightException()
A negative weight was encountered.

public NegativeWeightException(string message)
A negative weight was encountered.

message The error message that explains the reason for the exception.

public NegativeWeightException(string s, System.Exception exception)
A negative weight was encountered.

s The error message that explains the reason for the exception.

exception The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a
catch block that handles the inner exception.

A negative weight was encountered.

info The object that holds the serialized object data.

context The contextual information about the source or destination.

Methods

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.
GetObjectData

    virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
    When overridden in a derived class, sets the

ToString

    override public string ToString()
    Creates and returns a string representation of the current exception.

NewInitialGuessException Class

The iteration has not made good progress.

public class Imsl.Stat.NewInitialGuessException : IMSLException implements

Properties

HRESULT

    int HRESULT {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException

    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message

    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source

    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.
StackTrace
    public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

NewInitialGuessException
    public NewInitialGuessException()
    The iteration has not made good progress.

NewInitialGuessException
    public NewInitialGuessException(string message)
    The iteration has not made good progress.

        message    The error message that explains the reason for the exception.

NewInitialGuessException
    public NewInitialGuessException(string s, System.Exception exception)
    The iteration has not made good progress.

        s         The error message that explains the reason for the exception.

        exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

NewInitialGuessException
    The iteration has not made good progress.

        info  The object that holds the serialized object data.

        context The contextual information about the source or destination.

Methods

GetBaseException
    public System.Exception GetBaseException()
    When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.
GetObjectData
  virtual public void
  GetObjectData(System.Runtime.Serialization.SerializationInfo info,
  When overridden in a derived class, sets the

ToString
  override public string ToString()
  Creates and returns a string representation of the current exception.

---

**NoConvergenceException Class**

Convergence did not occur within the maximum number of iterations.

```csharp
public class Imsl.Stat.NoConvergenceException : IMSLEException implements
```

**Properties**

**HResult**

```csharp
int HResult {get; set; }
  Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
```

**HelpLink**

```csharp
virtual public string HelpLink {get; set; }
  Gets or sets a link to the help file associated with this exception.
```

**InnerException**

```csharp
public System.Exception InnerException {get; }
  Gets the System.Exception instance that caused the current exception.
```

**Message**

```csharp
virtual public string Message {get; }
  Gets a message that describes the current exception.
```

**Source**

```csharp
virtual public string Source {get; set; }
  Gets or sets the name of the application or the object that causes the error.
```
StackTrace

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current
exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

### Constructors

NoConvergenceException

```csharp
public NoConvergenceException(int maximumIterations)
```

Convergence did not occur within the maximum number of iterations.

- **maximumIterations** A `int` which specifies the maximum number of iterations allowed.

NoConvergenceException

```csharp
public NoConvergenceException()
```

Convergence did not occur within the maximum number of iterations.

NoConvergenceException

```csharp
public NoConvergenceException(string message)
```

Convergence did not occur within the maximum number of iterations.

- **message** The error message that explains the reason for the exception.

NoConvergenceException

```csharp
public NoConvergenceException(string s, System.Exception exception)
```

Convergence did not occur within the maximum number of iterations.

- **s** The error message that explains the reason for the exception.
- **exception** The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a
catch block that handles the inner exception.

NoConvergenceException

```csharp
```

Convergence did not occur within the maximum number of iterations.

- **info** The object that holds the serialized object data.
- **context** The contextual information about the source or destination.
Methods

GetBaseException
    virtual public System.Exception GetBaseException()
    When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData
    virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
    When overridden in a derived class, sets the

ToString
    override public string ToString()
    Creates and returns a string representation of the current exception.

NoDegreesOfFreedomException Class

No degrees of freedom error.

public class Imsl.Stat.NoDegreesOfFreedomException : IMSLEXception implements

Properties

HRESULT
    int HRESULT {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }

Exceptions
Gets a message that describes the current exception.

Source

```csharp
virtual public string Source { get; set; }
```

Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace { get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite { get; }
```

Gets the method that throws the current exception.

Constructors

**NoDegreesOfFreedomException**

```csharp
public NoDegreesOfFreedomException(int nvar, int nf)
```

No degrees of freedom error.

- **nvar** A int which specifies the number of variables.
- **nf** A int which specifies the number of factors.

**NoDegreesOfFreedomException**

```csharp
public NoDegreesOfFreedomException()
```

No degrees of freedom error.

**NoDegreesOfFreedomException**

```csharp
public NoDegreesOfFreedomException(string message)
```

No degrees of freedom error.

- **message** The error message that explains the reason for the exception.

**NoDegreesOfFreedomException**

```csharp
public NoDegreesOfFreedomException(string s, System.Exception exception)
```

No degrees of freedom error.

- **s** The error message that explains the reason for the exception.
- **exception** The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**NoDegreesOfFreedomException**

```csharp
```

No degrees of freedom error.

- **info** The object that holds the serialized object data.
- **context** The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

NoVariationInputException Class

There is no variation in the input data.

public class Imsl.Stat.NoVariationInputException : IMSLException implements

Properties

HResult

int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException

public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message

virtual public string Message {get; }
Gets a message that describes the current exception.

**Source**

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

**StackTrace**

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

**TargetSite**

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

### Constructors

**NoVariationInputException**

```csharp
public NoVariationInputException()
```

There is no variation in the input data.

**NoVariationInputException**

```csharp
public NoVariationInputException(string message)
```

There is no variation in the input data.

message  The error message that explains the reason for the exception.

**NoVariationInputException**

```csharp
public NoVariationInputException(string s, System.Exception exception)
```

There is no variation in the input data.

s  The error message that explains the reason for the exception.

exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**NoVariationInputException**

```csharp
```

There is no variation in the input data.

info  The object that holds the serialized object data.

context  The contextual information about the source or destination.
Methods

GetBaseException
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData
virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString
override public string ToString()
Creates and returns a string representation of the current exception.

NoVectorXException Class

No vector X satisfies all of the constraints.

public class Imsl.Stat.NoVectorXException : IMSLException implements

Properties

HRESULT
int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message
virtual public string Message {get; }
Gets a message that describes the current exception.

**Source**
```csharp
virtual public string Source {get; set; }
```
Gets or sets the name of the application or the object that causes the error.

**StackTrace**
```csharp
virtual public string StackTrace {get; }
```
Gets a string representation of the frames on the call stack at the time the current exception was thrown.

**TargetSite**
```csharp
public System.Reflection.MethodBase TargetSite {get; }
```
Gets the method that throws the current exception.

---

**Constructors**

**NoVectorXException**
```csharp
public NoVectorXException()
```
No vector X satisfies all of the constraints.

**NoVectorXException**
```csharp
public NoVectorXException(string message)
```
No vector X satisfies all of the constraints.

- **message**  The error message that explains the reason for the exception.

**NoVectorXException**
```csharp
public NoVectorXException(string s, System.Exception exception)
```
No vector X satisfies all of the constraints.

- **s**  The error message that explains the reason for the exception.
- **exception**  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**NoVectorXException**
```csharp
NoVectorXException(System.Runtime.Serialization.SerializationInfo info,
```
No vector X satisfies all of the constraints.

- **info**  The object that holds the serialized object data.
- **context**  The contextual information about the source or destination.
Methods

GetBaseException

```csharp
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.
```

GetObjectData

```csharp
virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the
```

ToString

```csharp
override public string ToString()
Creates and returns a string representation of the current exception.
```

NonPosVarianceException Class

The problem is ill-conditioned.

```csharp
public class Imsl.Stat.NonPosVarianceException : IMSLException implements
```

Properties

HResult

```csharp
int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
```

HelpLink

```csharp
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.
```

InnerException

```csharp
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.
```

Message

```csharp
virtual public string Message {get; }
```
Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

**Constructors**

NonPosVarianceException

```csharp
public NonPosVarianceException(double var)
```

The problem is ill-conditioned.

`var` A `double` which specifies the variance.

NonPosVarianceException

```csharp
public NonPosVarianceException()
```

The problem is ill-conditioned.

NonPosVarianceException

```csharp
public NonPosVarianceException(string message)
```

The problem is ill-conditioned.

`message` The error message that explains the reason for the exception.

NonPosVarianceException

```csharp
public NonPosVarianceException(string s, System.Exception exception)
```

The problem is ill-conditioned.

`s` The error message that explains the reason for the exception.

`exception` The exception that is the cause of the current exception. If the `innerException` parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

NonPosVarianceException

```csharp
```

The problem is ill-conditioned.

`info` The object that holds the serialized object data.

`context` The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

NonPosVarianceXYException Class

The problem is ill-conditioned.

public class Imsl.Stat.NonPosVarianceXYException : IMSLEException implements

Properties

HResult

int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException

public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message

virtual public string Message {get; }
Gets a message that describes the current exception.

**Source**

```
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

**StackTrace**

```
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

**TargetSite**

```
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

### Constructors

**NonPosVarianceXYException**

```
public NonPosVarianceXYException(string varName, double var)
```

The problem is ill-conditioned.

- **varName** A string which specifies either "X" or "Y".
- **var** A double which specifies the variance.

**NonPosVarianceXYException**

```
public NonPosVarianceXYException()
```

The problem is ill-conditioned.

**NonPosVarianceXYException**

```
public NonPosVarianceXYException(string message)
```

The problem is ill-conditioned.

- **message** The error message that explains the reason for the exception.

**NonPosVarianceXYException**

```
public NonPosVarianceXYException(string s, System.Exception exception)
```

The problem is ill-conditioned.

- **s** The error message that explains the reason for the exception.
- **exception** The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**NonPosVarianceXYException**

```
```

The problem is ill-conditioned.

- **info** The object that holds the serialized object data.
- **context** The contextual information about the source or destination.
Methods

GetBaseException

    virtual public System.Exception GetBaseException()
    When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

    When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

    override public string ToString()
    Creates and returns a string representation of the current exception.

NonPositiveEigenvalueException Class

Maximum number of iterations exceeded.

    public class Imsl.Stat.NonPositiveEigenvalueException : IMSLException
    implements

Properties

HResult

    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException

    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.
Message
    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

NonPositiveEigenvalueException
    public NonPositiveEigenvalueException(int iter, int i, double eval)
    Maximum number of iterations exceeded.

       iter   A int which specifies the iteration number.
       i      A int which specifies the eigenvalue index.
       eval   A double which specifies the eigenvalue.

NonPositiveEigenvalueException
    public NonPositiveEigenvalueException()
    Maximum number of iterations exceeded.

NonPositiveEigenvalueException
    public NonPositiveEigenvalueException(string message)
    Maximum number of iterations exceeded.

       message The error message that explains the reason for the exception.

NonPositiveEigenvalueException
    public NonPositiveEigenvalueException(string s, System.Exception exception)
    Maximum number of iterations exceeded.

       s      The error message that explains the reason for the exception.
       exception The exception that is the cause of the current exception. If the
                   innerException parameter is not a null reference, the current exception is raised in a
                   catch block that handles the inner exception.
NonPositiveEigenvalueException

Maximum number of iterations exceeded.

- **info**: The object that holds the serialized object data.
- **context**: The contextual information about the source or destination.

### Methods

#### GetBaseException

```csharp
virtual public System.Exception GetBaseException()
```
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

#### GetObjectData

```csharp
```
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

#### ToString

```csharp
override public string ToString()
```
Creates and returns a string representation of the current exception.

---

### NoPositiveVarianceException Class

No variable has positive variance. The Mahalanobis distances cannot be computed.

```csharp
public class Imsl.Stat.NoPositiveVarianceException : IMSLException implements
```

### Properties

#### HResult

```csharp
int HResult {get; set; }
```
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

NoPositiveVarianceException
    public NoPositiveVarianceException()
    No variable has positive variance. The Mahalanobis distances cannot be computed.

NoPositiveVarianceException
    public NoPositiveVarianceException(string message)
    No variable has positive variance. The Mahalanobis distances cannot be computed.

        message  The error message that explains the reason for the exception.

NoPositiveVarianceException
    public NoPositiveVarianceException(string s, System.Exception exception)
    No variable has positive variance. The Mahalanobis distances cannot be computed.

        s  The error message that explains the reason for the exception.

        exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
NoPositiveVarianceException


No variable has positive variance. The Mahalanobis distances cannot be computed.

info   The object that holds the serialized object data.
context  The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()

When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData


When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()

Creates and returns a string representation of the current exception.

NotCDFException Class

The function is not a Cumulative Distribution Function (CDF).

public class Imsl.Stat.NotCDFException : IMSLException implements

Properties

HRESULT

int HRESULT {get; set; }

Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink

    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException

    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message

    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source

    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace

    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

NotCDFException

    public NotCDFException(double lowerBound, double upperBound)
    The function is not a Cumulative Distribution Function (CDF).

        lowerBound        A double containing the lower bound to be displayed in message.
        upperBound       A double containing the upper bound to be displayed in message.

NotCDFException

    public NotCDFException(double range)
    The function is not a Cumulative Distribution Function (CDF).

        range        A double containing the probability of the range.

NotCDFException

    public NotCDFException(double x1, double x2, double f1)
    The function is not a Cumulative Distribution Function (CDF).

    The CDF function is not monotone, F(x1) = F(x2). No unique inverse exists.

        x1        is the first point
        x2        is the second point
\( f_1 \) is the common value for \( F(x_1) \) and \( F(x_2) \)

NotCDFException

```java
public NotCDFException(double lowerBound, double upperBound, double xx, int i)
```

The function is not a Cumulative Distribution Function (CDF).

The cdf function is not a cumulative distribution function because its value at a cutpoint is out of the expected range, \([p_{lower}, p_{upper}]\).

- `lowerBound` A `double` containing the lower bound for the CDF value.
- `upperBound` A `double` containing the upper bound for the CDF value.
- `xx` A `double` containing the value at a cutpoint.
- `i` The index of the cutpoint that is out of range.

NotCDFException

```java
public NotCDFException()
```

The function is not a Cumulative Distribution Function (CDF).

NotCDFException

```java
public NotCDFException(string message)
```

The function is not a Cumulative Distribution Function (CDF).

- `message` The error message that explains the reason for the exception.

NotCDFException

```java
public NotCDFException(string s, System.Exception exception)
```

The function is not a Cumulative Distribution Function (CDF).

- `s` The error message that explains the reason for the exception.
- `exception` The exception that is the cause of the current exception. If the `innerException` parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

NotCDFException

```java
NotCDFException(System.Runtime.Serialization.SerializationInfo info,
```

The function is not a Cumulative Distribution Function (CDF).

- `info` The object that holds the serialized object data.
- `context` The contextual information about the source or destination.

**Methods**

GetBaseException

```java
virtual public System.Exception GetBaseException()
```

When overridden in a derived class, returns the `System.Exception` that is the root cause of one or more subsequent exceptions.
GetObjectData
    virtual public void
    GetObjectData(System.Runtime.Serialization.SerializationInfo info,
    When overridden in a derived class, sets the

ToString
    override public string ToString()
    Creates and returns a string representation of the current exception.

---

**NotPositiveDefiniteException Class**

Covariance matrix is not positive definite.

public class Imsl.Stat.NotPositiveDefiniteException : IMSLException implements

**Properties**

HRESULT
    int HRESULT {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.
StackTrace

```csharp
virtual public string StackTrace { get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite { get; }
```

Gets the method that throws the current exception.

## Constructors

**NotPositiveDefiniteException**

```csharp
public NotPositiveDefiniteException(int i)
```

Covariance matrix is not positive definite.

- **i**  Variable i is linearly related to the other variables in the factor analysis.

**NotPositiveDefiniteException**

```csharp
public NotPositiveDefiniteException()
```

Covariance matrix is not positive definite.

**NotPositiveDefiniteException**

```csharp
public NotPositiveDefiniteException(string message)
```

Covariance matrix is not positive definite.

- **message**  The error message that explains the reason for the exception.

**NotPositiveDefiniteException**

```csharp
public NotPositiveDefiniteException(string s, System.Exception exception)
```

Covariance matrix is not positive definite.

- **s**  The error message that explains the reason for the exception.
- **exception**  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**NotPositiveDefiniteException**

```csharp
```

Covariance matrix is not positive definite.

- **info**  The object that holds the serialized object data.
- **context**  The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

NotPositiveSemiDefiniteException Class

Covariance matrix is not positive semi-definite.

public class Imsl.Stat.NotPositiveSemiDefiniteException : IMSLException
implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException

public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.
Message
   virtual public string Message {get; }
   Gets a message that describes the current exception.

Source
   virtual public string Source {get; set; }
   Gets or sets the name of the application or the object that causes the error.

StackTrace
   virtual public string StackTrace {get; }
   Gets a string representation of the frames on the call stack at the time the current
   exception was thrown.

TargetSite
   public System.Reflection.MethodBase TargetSite {get; }
   Gets the method that throws the current exception.

Constructors

NotPositiveSemiDefiniteException
   public NotPositiveSemiDefiniteException()
   Covariance matrix is not positive semi-definite.

NotPositiveSemiDefiniteException
   public NotPositiveSemiDefiniteException(string message)
   Covariance matrix is not positive semi-definite.

message  The error message that explains the reason for the exception.

NotPositiveSemiDefiniteException
   public NotPositiveSemiDefiniteException(string s, System.Exception
   exception)
   Covariance matrix is not positive semi-definite.

s        The error message that explains the reason for the exception.

exception  The exception that is the cause of the current exception. If the
   innerException parameter is not a null reference, the current exception is raised in a
   catch block that handles the inner exception.

NotPositiveSemiDefiniteException
   NotPositiveSemiDefiniteException(System.Runtime.Serialization.SerializationInfo
   Covariance matrix is not positive semi-definite.

info   The object that holds the serialized object data.

context  The contextual information about the source or destination.
**Methods**

GetBaseException

```csharp
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.
```

GetObjectData

```csharp
virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the
```

ToString

```csharp
override public string ToString()
Creates and returns a string representation of the current exception.
```

---

**NotSemiDefiniteException Class**

Hessian matrix is not semi-definite.

```csharp
public class Imsl.Stat.NotSemiDefiniteException : IMSLException implements
```

**Properties**

HResult

```csharp
int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
```

HelpLink

```csharp
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.
```

InnerException

```csharp
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.
```

Message

```csharp
virtual public string Message {get; }
```
Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```
Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```
Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```
Gets the method that throws the current exception.

Constructors

NotSemiDefiniteException

```csharp
public NotSemiDefiniteException()
Hessian matrix is not semi-definite.
```

NotSemiDefiniteException

```csharp
public NotSemiDefiniteException(string message)
Hessian matrix is not semi-definite.
```
message The error message that explains the reason for the exception.

NotSemiDefiniteException

```csharp
public NotSemiDefiniteException(string s, System.Exception exception)
Hessian matrix is not semi-definite.
```
s The error message that explains the reason for the exception.
exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

NotSemiDefiniteException

```csharp
Hessian matrix is not semi-definite.
```
info The object that holds the serialized object data.
context The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

NoVariablesEnteredException Class

No Variables can enter the model.

public class Imsl.Stat.NoVariablesEnteredException : IMSLException implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException

public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message

virtual public string Message {get; }
Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

### Constructors

**NoVariablesEnteredException**

```csharp
public NoVariablesEnteredException()
```

No Variables can enter the model.

```csharp
public NoVariablesEnteredException(string message)
```

No Variables can enter the model.

- **message** The error message that explains the reason for the exception.

```csharp
public NoVariablesEnteredException(string s, System.Exception exception)
```

No Variables can enter the model.

- **s** The error message that explains the reason for the exception.
- **exception** The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

```csharp
```

No Variables can enter the model.

- **info** The object that holds the serialized object data.
- **context** The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

NoVariablesException Class

No variables can enter the model.

public class Imsl.Stat.NoVariablesException : IMSLException implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException

public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message

virtual public string Message {get; }
Gets a message that describes the current exception.

**Source**

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

**StackTrace**

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

**TargetSite**

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

### Constructors

**NoVariablesException**

```csharp
public NoVariablesException()
```

No variables can enter the model.

**NoVariablesException**

```csharp
public NoVariablesException(string message)
```

No variables can enter the model.

- **message** The error message that explains the reason for the exception.

**NoVariablesException**

```csharp
public NoVariablesException(string s, System.Exception exception)
```

No variables can enter the model.

- **s** The error message that explains the reason for the exception.
- **exception** The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**NoVariablesException**

```csharp
```

No variables can enter the model.

- **info** The object that holds the serialized object data.
- **context** The contextual information about the source or destination.
Methods

GetBaseException
    virtual public System.Exception GetBaseException()
    When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData
    virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
    When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString
    override public string ToString()
    Creates and returns a string representation of the current exception.

NoVariationInputException Class

There is no variation in the input data.

public class Imsl.Stat.NoVariationInputException : IMSLException implements

Properties

HRESULT
    int HRESULT {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }
Gets a message that describes the current exception.

Source
virtual public string Source {get; set; }
Gets or sets the name of the application or the object that causes the error.

StackTrace
virtual public string StackTrace {get; }
Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
public System.Reflection.MethodBase TargetSite {get; }
Gets the method that throws the current exception.

Constructors

NoVariationInputException
public NoVariationInputException()
There is no variation in the input data.

NoVariationInputException
public NoVariationInputException(string message)
There is no variation in the input data.

message The error message that explains the reason for the exception.

NoVariationInputException
public NoVariationInputException(string s, System.Exception exception)
There is no variation in the input data.

s The error message that explains the reason for the exception.

exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

NoVariationInputException
There is no variation in the input data.

info The object that holds the serialized object data.
context The contextual information about the source or destination.
Methods

GetBaseException

```csharp
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.
```

GetObjectData

```csharp
virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.
```

ToString

```csharp
override public string ToString()
Creates and returns a string representation of the current exception.
```

NoVectorXException Class

No vector X satisfies all of the constraints.

```csharp
public class Imsl.Stat.NoVectorXException : IMSLException implements Properties
```

Properties

HResult

```csharp
    int HResult {get; set; }
    Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
```

HelpLink

```csharp
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.
```

InnerException

```csharp
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.
```

Message

```csharp
    virtual public string Message {get; }
```
Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```
Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```
Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```
Gets the method that throws the current exception.

Constructors

NoVectorXException

```csharp
public NoVectorXException()
No vector X satisfies all of the constraints.
```

NoVectorXException

```csharp
public NoVectorXException(string message)
No vector X satisfies all of the constraints.
```

```csharp
message The error message that explains the reason for the exception.
```

NoVectorXException

```csharp
public NoVectorXException(string s, System.Exception exception)
No vector X satisfies all of the constraints.
```

```csharp
s The error message that explains the reason for the exception.
```

```csharp
exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
```

NoVectorXException

```csharp
No vector X satisfies all of the constraints.
```

```csharp
info The object that holds the serialized object data.
```

```csharp
context The contextual information about the source or destination.
```
Methods

GetBaseException
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData
virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString
override public string ToString()
Creates and returns a string representation of the current exception.

PooledCovarianceSingularException Class

The pooled variance-Covariance matrix is singular.

public class Imsl.Stat.PooledCovarianceSingularException : IMSLException
implements

Properties

HRESULT
int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.
Message
virtual public string Message {get; }
Gets a message that describes the current exception.

Source
virtual public string Source {get; set; }
Gets or sets the name of the application or the object that causes the error.

StackTrace
virtual public string StackTrace {get; }
Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
public System.Reflection.MethodBase TargetSite {get; }
Gets the method that throws the current exception.

Constructors

PooledCovarianceSingularException
  public PooledCovarianceSingularException()
  The pooled variance-Covariance matrix is singular.

PooledCovarianceSingularException
  public PooledCovarianceSingularException(string message)
  The pooled variance-Covariance matrix is singular.
  message  The error message that explains the reason for the exception.

PooledCovarianceSingularException
  public PooledCovarianceSingularException(string s, System.Exception exception)
  The pooled variance-Covariance matrix is singular.
  s  The error message that explains the reason for the exception.
  exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

PooledCovarianceSingularException
  The pooled variance-Covariance matrix is singular.
  info  The object that holds the serialized object data.
  context  The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

RankException Class

Rank of covariance matrix error.

class Imsl.Stat.RankException : IMSLException implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException

public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message

virtual public string Message {get; }
Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

**Constructors**

**RankException**

```csharp
public RankException(int rank, int nf)
```

Rank of covariance matrix error.

- **rank** A `int` which specifies the rank of the covariance matrix.
- **nf** A `int` which specifies the number of factors.

**RankException**

```csharp
public RankException()
```

Rank of covariance matrix error.

**RankException**

```csharp
public RankException(string message)
```

Rank of covariance matrix error.

- **message** The error message that explains the reason for the exception.

**RankException**

```csharp
public RankException(string s, System.Exception exception)
```

Rank of covariance matrix error.

- **s** The error message that explains the reason for the exception.
- **exception** The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**RankException**

```csharp
RankException(System.Runtime.Serialization.SerializationInfo info,
```

Rank of covariance matrix error.

- **info** The object that holds the serialized object data.
- **context** The contextual information about the source or destination.
Methods

GetBaseException

```
virtual public System.Exception GetBaseException()
```
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

```
virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
```
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

```
override public string ToString()
```
Creates and returns a string representation of the current exception.

ScaleFactorZeroException Class

The computations cannot continue because a scale factor is zero.

```
public class Imsl.Stat.ScaleFactorZeroException : IMSLException implements
```

Properties

HResult

```
int HResult {get; set; }
```
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

```
virtual public string HelpLink {get; set; }
```
Gets or sets a link to the help file associated with this exception.

InnerException

```
public System.Exception InnerException {get; }
```
Gets the System.Exception instance that caused the current exception.

Message

```
virtual public string Message {get; }
```
Gets a message that describes the current exception.

**Source**

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

**StackTrace**

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

**TargetSite**

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

### Constructors

**ScaleFactorZeroException**

```csharp
public ScaleFactorZeroException(int index)
```

The computations cannot continue because a scale factor is zero.

- **index** — An `int` which specifies the index of the scale factor array at which scale factor is zero.

**ScaleFactorZeroException**

```csharp
public ScaleFactorZeroException()
```

The computations cannot continue because a scale factor is zero.

**ScaleFactorZeroException**

```csharp
public ScaleFactorZeroException(string message)
```

The computations cannot continue because a scale factor is zero.

- **message** — The error message that explains the reason for the exception.

**ScaleFactorZeroException**

```csharp
public ScaleFactorZeroException(string s, System.Exception exception)
```

The computations cannot continue because a scale factor is zero.

- **s** — The error message that explains the reason for the exception.
- **exception** — The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**ScaleFactorZeroException**

```csharp
```

The computations cannot continue because a scale factor is zero.

- **info** — The object that holds the serialized object data.
- **context** — The contextual information about the source or destination.
Methods

GetBaseException
  virtual public System.Exception GetBaseException()
  When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData
  virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
  When overridden in a derived class, sets the

ToString
  override public string ToString()
  Creates and returns a string representation of the current exception.

SingularException Class

Covariance matrix is singular.

public class Imsl.Stat.SingularException : IMSLException implements

Properties

HRESULT
  int HRESULT {get; set; }
  Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
  virtual public string HelpLink {get; set; }
  Gets or sets a link to the help file associated with this exception.

InnerException
  public System.Exception InnerException {get; }
  Gets the System.Exception instance that caused the current exception.

Message
  virtual public string Message {get; }
Gets a message that describes the current exception.

**Source**

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

**StackTrace**

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

**TargetSite**

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

### Constructors

**SingularException**

```csharp
public SingularException(int i)
```

Covariance matrix is singular.

- **i**  Variable i is linearly related to the other variables.

**SingularException**

```csharp
public SingularException()
```

Covariance matrix is singular.

**SingularException**

```csharp
public SingularException(string message)
```

Covariance matrix is singular.

- **message**  The error message that explains the reason for the exception.

**SingularException**

```csharp
public SingularException(string s, System.Exception exception)
```

Covariance matrix is singular.

- **s**  The error message that explains the reason for the exception.
- **exception**  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**SingularException**

```csharp
```

Covariance matrix is singular.

- **info**  The object that holds the serialized object data.
- **context**  The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

SumOfWeightsNegException Class

The sum of the weights have become negative.

public class Imsl.Stat.SumOfWeightsNegException : IMSLEException implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException

public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message

virtual public string Message {get; }
Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

### Constructors

**SumOfWeightsNegException**

```csharp
public SumOfWeightsNegException(int group)
```

The sum of the weights have become negative.

- `group` A `int` which specifies the group for which the sum of the weights have become negative.

**SumOfWeightsNegException**

```csharp
public SumOfWeightsNegException()
```

The sum of the weights have become negative.

**SumOfWeightsNegException**

```csharp
public SumOfWeightsNegException(string message)
```

The sum of the weights have become negative.

- `message` The error message that explains the reason for the exception.

**SumOfWeightsNegException**

```csharp
public SumOfWeightsNegException(string s, System.Exception exception)
```

The sum of the weights have become negative.

- `s` The error message that explains the reason for the exception.
- `exception` The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

**SumOfWeightsNegException**

```csharp
public SumOfWeightsNegException(System.Runtime.Serialization.SerializationInfo info,
```

The sum of the weights have become negative.

- `info` The object that holds the serialized object data.
- `context` The contextual information about the source or destination.
Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

TooManyCallsException Class

The number of calls to the function has exceeded the maximum number of iterations.

public class Imsl.Stat.TooManyCallsException : IMSLEException implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink

virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException

public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message

virtual public string Message {get; }
Gets a message that describes the current exception.

Source

```csharp
virtual public string Source {get; set; }
```

Gets or sets the name of the application or the object that causes the error.

StackTrace

```csharp
virtual public string StackTrace {get; }
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite

```csharp
public System.Reflection.MethodBase TargetSite {get; }
```

Gets the method that throws the current exception.

Constructors

TooManyCallsException

```csharp
public TooManyCallsException()
```

The number of calls to the function has exceeded the maximum number of iterations.

TooManyCallsException

```csharp
public TooManyCallsException(string message)
```

The number of calls to the function has exceeded the maximum number of iterations.

```csharp
message  The error message that explains the reason for the exception.
```

TooManyCallsException

```csharp
public TooManyCallsException(string s, System.Exception exception)
```

The number of calls to the function has exceeded the maximum number of iterations.

```csharp
s  The error message that explains the reason for the exception.

exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
```

TooManyCallsException

```csharp
TooManyCallsException(System.Runtime.Serialization.SerializationInfo info,
```

The number of calls to the function has exceeded the maximum number of iterations.

```csharp
info  The object that holds the serialized object data.

context  The contextual information about the source or destination.
```
Methods

GetBaseException

\begin{verbatim}
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.
\end{verbatim}

GetObjectData

\begin{verbatim}
virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the
\end{verbatim}

ToString

\begin{verbatim}
override public string ToString()
Creates and returns a string representation of the current exception.
\end{verbatim}

TooManyFunctionEvaluationsException Class

Maximum number of function evaluations exceeded.

\begin{verbatim}
public class Imsl.Stat.TooManyFunctionEvaluationsException : IMSLException
implements
\end{verbatim}

Properties

HResult

\begin{verbatim}
int HResult {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
\end{verbatim}

HelpLink

\begin{verbatim}
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.
\end{verbatim}

InnerException

\begin{verbatim}
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.
\end{verbatim}
Message
    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

TooManyFunctionEvaluationsException
    public TooManyFunctionEvaluationsException(int maximumNumberOfEvaluations)
    Maximum number of function evaluations exceeded.

    maximumNumberOfEvaluations A int which specifies the maximum number of function evaluations allowed.

TooManyFunctionEvaluationsException
    public TooManyFunctionEvaluationsException()
    Maximum number of function evaluations exceeded.

TooManyFunctionEvaluationsException
    public TooManyFunctionEvaluationsException(string message)
    Maximum number of function evaluations exceeded.

    message The error message that explains the reason for the exception.

TooManyFunctionEvaluationsException
    public TooManyFunctionEvaluationsException(string s, System.Exception exception)
    Maximum number of function evaluations exceeded.

    s The error message that explains the reason for the exception.

    exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
TooManyFunctionEvaluationsException

Maximum number of function evaluations exceeded.

info  The object that holds the serialized object data.
context  The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.

TooManyIterationsException Class

Maximum number of iterations exceeded.

public class Imsl.Stat.TooManyIterationsException : IMSLEexception implements

Properties

HRESULT

int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink
  virtual public string HelpLink {get; set; }
  Gets or sets a link to the help file associated with this exception.

InnerException
  public System.Exception InnerException {get; }
  Gets the System.Exception instance that caused the current exception.

Message
  virtual public string Message {get; }
  Gets a message that describes the current exception.

Source
  virtual public string Source {get; set; }
  Gets or sets the name of the application or the object that causes the error.

StackTrace
  virtual public string StackTrace {get; }
  Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
  public System.Reflection.MethodBase TargetSite {get; }
  Gets the method that throws the current exception.

Constructors

TooManyIterationsException
  public TooManyIterationsException(int maximumNumberOfIterations)
  Maximum number of iterations exceeded.

    maximumNumberOfIterations A int which specifies the maximum number of iterations allowed.

TooManyIterationsException
  public TooManyIterationsException()
  Maximum number of iterations exceeded.

TooManyIterationsException
  public TooManyIterationsException(string message)
  Maximum number of iterations exceeded.

    message The error message that explains the reason for the exception.

TooManyIterationsException
  public TooManyIterationsException(string s, System.Exception exception)
  Maximum number of iterations exceeded.

    s The error message that explains the reason for the exception.
exception The exception that is the cause of the current exception. If the
innerException parameter is not a null reference, the current exception is raised in a
catch block that handles the inner exception.

TooManyIterationsException
ToTooManyIterationsException(System.Runtime.Serialization.SerializationInfo
Maximum number of iterations exceeded.

info The object that holds the serialized object data.
context The contextual information about the source or destination.

Methods

GetBaseException
virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause
of one or more subsequent exceptions.

GetObjectData
virtual public void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the

ToString
override public string ToString()
Creates and returns a string representation of the current exception.

TooManyJacobianEvalException Class

Maximum number of Jacobian evaluations exceeded.

public class Imsl.Stat.TooManyJacobianEvalException : IMSLExccepion implements

Properties

HRESULT
int HRESULT {get; set; }
Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

HelpLink
virtual public string HelpLink {get; set; }
Gets or sets a link to the help file associated with this exception.

InnerException
public System.Exception InnerException {get; }
Gets the System.Exception instance that caused the current exception.

Message
virtual public string Message {get; }
Gets a message that describes the current exception.

Source
virtual public string Source {get; set; }
Gets or sets the name of the application or the object that causes the error.

StackTrace
virtual public string StackTrace {get; }
Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
public System.Reflection.MethodBase TargetSite {get; }
Gets the method that throws the current exception.

Constructors

TooManyJacobianEvalException
public TooManyJacobianEvalException()
Maximum number of Jacobian evaluations exceeded.

TooManyJacobianEvalException
public TooManyJacobianEvalException(string message)
Maximum number of Jacobian evaluations exceeded.

message  The error message that explains the reason for the exception.

TooManyJacobianEvalException
public TooManyJacobianEvalException(string s, System.Exception exception)
Maximum number of Jacobian evaluations exceeded.

s  The error message that explains the reason for the exception.

exception  The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
TooManyJacobianEvalException


Maximum number of Jacobian evaluations exceeded.

- **info** The object that holds the serialized object data.
- **context** The contextual information about the source or destination.

**Methods**

- **GetBaseException**
  
  virtual public System.Exception GetBaseException()
  
  When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

- **GetObjectData**
  
  
  When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

- **ToString**
  
  override public string ToString()
  
  Creates and returns a string representation of the current exception.

**TooManyObsDeletedException Class**

More observations have been deleted than were originally entered (the sum of frequencies has become negative).

public class Imsl.Stat.TooManyObsDeletedException : IMSLException implements

**Properties**

- **HResult**
  
  int HResult {get; set; }
  
  Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink
    
    virtual public string HelpLink {get; set; }
    
    Gets or sets a link to the help file associated with this exception.

InnerException
    
    public System.Exception InnerException {get; }
    
    Gets the System.Exception instance that caused the current exception.

Message
    
    virtual public string Message {get; }
    
    Gets a message that describes the current exception.

Source
    
    virtual public string Source {get; set; }
    
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    
    virtual public stringStackTrace {get; }
    
    Gets a string representation of the frames on the call stack at the time the current exception was thrown.

TargetSite
    
    public System.Reflection.MethodBase TargetSite {get; }
    
    Gets the method that throws the current exception.

Constructors

TooManyObsDeletedException
    
    public TooManyObsDeletedException()
    
    More observations have been deleted than were originally entered (the sum of frequencies has become negative).

TooManyObsDeletedException
    
    public TooManyObsDeletedException(string message)
    
    More observations have been deleted than were originally entered (the sum of frequencies has become negative).

    message The error message that explains the reason for the exception.

TooManyObsDeletedException
    
    public TooManyObsDeletedException(string s, System.Exception exception)
    
    More observations have been deleted than were originally entered (the sum of frequencies has become negative).

    s The error message that explains the reason for the exception.

    exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.
TooManyObsDeletedException


More observations have been deleted than were originally entered (the sum of frequencies has become negative).

- **info**  The object that holds the serialized object data.
- **context**  The contextual information about the source or destination.

**Methods**

GetBaseException

```
virtual public System.Exception GetBaseException()
```

When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

```
```

When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

```
override public string ToString()
```

Creates and returns a string representation of the current exception.

---

**VarsDeterminedException Class**

The variables are determined by the equality constraints.

```
public class Imsl.Stat.VarsDeterminedException : IMSLException implements

Properties

- **HResult**

```
int HResult {get; set; }
```

Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink
    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException
    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message
    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source
    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace
    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.

TargetSite
    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

VarsDeterminedException
    public VarsDeterminedException()
    The variables are determined by the equality constraints.

VarsDeterminedException
    public VarsDeterminedException(string message)
    The variables are determined by the equality constraints.

        message  The error message that explains the reason for the exception.

VarsDeterminedException
    public VarsDeterminedException(string s, System.Exception exception)
    The variables are determined by the equality constraints.

        s  The error message that explains the reason for the exception.

        exception  The exception that is the cause of the current exception. If the
                    innerException parameter is not a null reference, the current exception is raised in a
                    catch block that handles the inner exception.
VarsDeterminedException

The variables are determined by the equality constraints.

- **info**: The object that holds the serialized object data.
- **context**: The contextual information about the source or destination.

**Methods**

- **GetBaseException**
  
  virtual public System.Exception GetBaseException()
  When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

- **GetObjectData**
  
  When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

- **ToString**
  
  override public string ToString()
  Creates and returns a string representation of the current exception.

---

**ZeroNormException Class**

The computations cannot continue because the Euclidean norm of the column is equal to zero.

```csharp
public class Imsl.Stat.ZeroNormException : IMSLException implements
```

**Properties**

- **HRESULT**
  
  int HRESULT {get; set; }
  Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
HelpLink

    virtual public string HelpLink {get; set; }
    Gets or sets a link to the help file associated with this exception.

InnerException

    public System.Exception InnerException {get; }
    Gets the System.Exception instance that caused the current exception.

Message

    virtual public string Message {get; }
    Gets a message that describes the current exception.

Source

    virtual public string Source {get; set; }
    Gets or sets the name of the application or the object that causes the error.

StackTrace

    virtual public string StackTrace {get; }
    Gets a string representation of the frames on the call stack at the time the current
    exception was thrown.

TargetSite

    public System.Reflection.MethodBase TargetSite {get; }
    Gets the method that throws the current exception.

Constructors

ZeroNormException

    public ZeroNormException(int index)
    The computations cannot continue because the Euclidean norm of the column is equal to
    zero.

    index An int which specifies the column index for which the norm has been found to
    be zero.

ZeroNormException

    public ZeroNormException()
    The computations cannot continue because the Euclidean norm of the column is equal to
    zero.

ZeroNormException

    public ZeroNormException(string message)
    The computations cannot continue because the Euclidean norm of the column is equal to
    zero.

    message The error message that explains the reason for the exception.
ZeroNormException

public ZeroNormException(string s, System.Exception exception)
The computations cannot continue because the Euclidean norm of the column is equal to zero.

s The error message that explains the reason for the exception.

exception The exception that is the cause of the current exception. If the innerException parameter is not a null reference, the current exception is raised in a catch block that handles the inner exception.

ZeroNormException

ZeroNormException(System.Runtime.Serialization.SerializationInfo info,
The computations cannot continue because the Euclidean norm of the column is equal to zero.

info The object that holds the serialized object data.

context The contextual information about the source or destination.

Methods

GetBaseException

virtual public System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

GetObjectData

virtual public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.

ToString

override public string ToString()
Creates and returns a string representation of the current exception.
Chapter 26: References

References

Abramowitz and Stegun

Ahrens and Dieter

Akima

Ashcraft

Ashcraft et al.

Atkinson (1979)
Atkinson, A.C. (1979), A family of switching algorithms for the computer generation of beta random variates, Biometrika, 66, 141-145.

Atkinson (1978)

945
Barnett

Barrett and Heal

Bays and Durham

Blom

Boisvert

Bosten and Battiste

Brent

Brigham

Burgoyne

Carlson

Carlson and Notis

Carlson and Foley

Cheng


Cohen and Taylor


Cooley and Tukey


Cooper


Courant and Hilbert


Craven and Wahba


Crowe et al.


Davis and Rabinowitz


de Boor


Dennis and Schnabel


Dongarra et al.


Draper and Smith

References

**DuCroz et al.**


**Duff et al.**


**Duff and Reid**


**Enright and Pryce**


**Farebrother and Berry**


**Fisher**


**Fishman and Moore**


**Forsythe**


**Franke**


**Garbow et al.**


Garbow, B.S., G. Giunta, J.N. Lyness, and A. Murli (1988), Software for an implementation of

Gautschi

Gear

Gentleman

George and Liu

Gill and Murray

Gill et al.

Goldfarb and Idnani

Golub

Golub and Van Loan


Golub and Welsch

Gregory and Karney

References

**Griffin and Redfish**


**Grosse**


**Guerra and Tapia**


**Hageman and Young**


**Hanson**


**Hardy**


**Hart et al.**


**Healy**


**Herraman**


**Higham**


**Hill**

**Hindmarsh**


**Hinkley**


**Huber**


**Hull et al.**


**Irvine et al.**


**Jackson et al.**


**Jenkins**


**Jenkins and Traub**


**Jöhnk**


**Kendall and Stuart**

Kennedy and Gentle

Kernighan and Richtie

Kinnucan and Kuki

Kirk

Knuth

Learmonth and Lewis

Lehmann

Levenberg

Leavenworth

Lewis et al.

Liepman

Liu


**Lyness and Giunta**


**Madsen and Sincovec**


**Maindonald**


**Marquardt**


**Martin and Wilkinson**


**Mayle**


**Michelli**


**Michelli et al.**

Micchelli, C.A., T.J. Rivlin, and S. Winograd (1976), The optimal recovery of smooth


**Microsoft Excel User Education Team**


**Moler and Stewart**


**Moré et al.**


**Müller**


**Murtagh**


**Murty**


**Neter and Wasserman**


**Neter et al.**


**Osterby and Zlatev**


**Owen**


---

**IMSL C# Numerical Library**
Parlett

Petro

Piessens et al.

Powell

Reinsch

Rice

Saad and Schultz

Sallas and Lionti

Savage
Schittkowski


Schmeiser

Schmeiser and Babu

Schmeiser and Kachitvichyanukul

Schmeiser and Lal

Seidler and Carmichael

Shampine

Shampine and Gear

Sincocke and Madsen

**Singleton**


**Smith et al.**


**Smith**


**Stewart**


**Stoer**


**Strecok**


**Stroud and Secrest**


**Temme**


**Tezuka**


**Thompson and Barnett**

Thompson, I.J. and A.R. Barnett (1987), Modified Bessel functions $I_{n}(z)$ and $K_{n}(z)$ of real order and complex argument, *Computer Physics Communication*, 47, 245-257.

**Tukey**


**Velleman and Hoaglin**

References

**Walker**


**Watkins**


**Weeks**

# Index

AllDeletedException, 840  
AllMissingException, 842  
ANOVA, 341  
ANOVAFactorial, 346  
ANOVAFactorial.ErrorCalculation, 356  
ARMA, 446  
ARMA.ParamEstimation, 464  
AutoCorrelation, 415  
AutoCorrelation.StdErr, 423  
BadInitialGuessException, 779  
BadVarianceException, 844  
Bessel, 187  
Bond, 624  
Bond.Frequency, 656  
BoundedLeastSquares, 152  
BoundedLeastSquares.IFunction, 158  
BoundedLeastSquares.IJacobian, 159  
BoundsInconsistentException, 781  
BsInterpolate, 60  
BsLeastSquares, 63  
BSpline, 57  
CategoricalGenLinModel, 372  
CategoricalGenLinModel.DistributionParameterModel, 390  
Cdf, 551  
ChiSquaredTest, 403  
Cholesky, 16  
ClassificationVariableException, 846  
ClassificationVariableLimitException, 848  
ClassificationVariableValueException, 850  
ClusterHierarchical, 499  
ClusterKMeans, 487  
ClusterNoPointsException, 852  
Complex, 193  
ComplexFFT, 98  
ComplexLU, 13  
ComplexMatrix, 6  
ConstraintEvaluationException, 783  
ConstraintsInconsistentException, 785  
ConstraintsNotSatisfiedException, 787  
ConstrInconsistentException, 854  
ContingencyTable, 359  
Covariances, 246  
Covariances.MatrixType, 251  
CovarianceSparseException, 856  
CrossCorrelation, 424  
CrossCorrelation.StdErr, 435  
CsAkima, 39  
CsInterpolate, 41  
CsInterpolate.Condition, 44  
CsPeriodic, 45  
CsShape, 47  
CsSmooth, 50  
CsSmoothC2, 53  
CyclingIsOccurringException, 858  
DayCountBasis, 657  
DeleteObservationsException, 860  
DidNotConvergeException, 789, 862  
DiffObsDeletedException, 864  
DiscriminantAnalysis, 525  
DiscriminantAnalysis.Classification, 546  
DiscriminantAnalysis.CovarianceMatrix, 546  
DiscriminantAnalysis.Discrimination, 545  
DiscriminantAnalysis.PriorProbabilities, 547  
Dissimilarities, 496  
Eigen, 29  
EigenvalueException, 866  
EmptyGroupException, 868  
EpochGroupException, 869  
EpochTrainer, 719
EpsilonAlgorithm, 222
EqConstrInconsistentException, 870
EqualityConstraintsException, 791
FactorAnalysis, 507
FactorAnalysis.MatrixType, 523
FactorAnalysis.Model, 524
FalseConvergenceException, 793
FaureSequence, 588
FeedForwardNetwork, 688
FFT, 94
Finance, 594
Finance.Period, 623

GARCH, 469
HiddenLayer, 701
HyperRectangleQuadrature, 82
HyperRectangleQuadrature.IFunction, 85
IActivation, 708
IBasisPart, 623
ICdfFunction, 572
IIlConditionedException, 795, 872
IMSLException, 773
InconsistentSystemException, 797
IncreaseErrRelException, 874
InputLayer, 700
InputNode, 705
InverseCdf, 572
IRandomSequence, 592
IRegressionBasis, 339
ITrainer, 710

KalmanFilter, 474
Layer, 699
LeastSquaresTrainer, 716
LimitingAccuracyException, 799
LinearlyDependentGradientsException, 801
LinearProgramming, 135
LinearRegression, 297
LinearRegression.CoefficientTTestsValue, 301
Link, 709
LU, 10

Matrix, 3
MatrixSingularException, 876
MaxIterationsException, 803
MaxNumberStepsAllowedException, 805
MinConGenLin, 144
MinConGenLin.IFunction, 151
MinConGenLin.IGradient, 151
MinConNLP, 159
MinConNLP.IFunction, 168
MinConNLP.IGradient, 168
MinUncon, 117
MinUncon.IDerivative, 122
MinUncon.IFunction, 122
MinUnconMultiVar, 122
MinUnconMultiVar.IFunction, 128
MinUnconMultiVar.IGradient, 128
MoreObsDelThanEnteredException, 878
MultiCrossCorrelation, 435
MultipleComparisons, 357
NegativeFreqException, 881
NegativeWeightException, 883
Network, 762
NewInitialGuessException, 885
NoAcceptableStepsizeException, 807
NoConvergenceException, 887
Node, 704
NoDegreesOfFreedomException, 889
NonlinearRegression, 302
NonlinearRegression.IDerivative, 314
NonlinearRegression.IFunction, 315
NonlinLeastSquares, 129
NonlinLeastSquares.IFunction, 135
NonlinLeastSquares.IJacobian, 135
NonPositiveEigenvalueException, 899
NonPosVarianceException, 895
NonPosVarianceXYException, 897
NoPositiveVarianceException, 901
NormalityTest, 408
NormOneSample, 251
NormTwoSample, 256
NotCDFException, 903
NotPositiveDefiniteException, 906
NotPositiveSemiDefiniteException, 908
NotSemiDefiniteException, 910
NotSPDException, 809
NoVariablesEnteredException, 912
NoVariablesException, 914
NoVariationInputException, 891, 916
NoVectorXException, 893, 918
NumericDifficultyException, 811
ObjectiveEvaluationException, 813
OdeRungeKutta, 88
OdeRungeKutta.IFunction, 91
OutputLayer, 702
OutputPerceptron, 707
PenaltyFunctionPointInfeasibleException, 815
Perceptron, 706
Physical, 211
PooledCovarianceSingularException, 920
PrintMatrix, 225
PrintMatrix.MatrixType, 232
PrintMatrixFormat, 229
PrintMatrixFormat.ColumnLabelType, 234
PrintMatrixFormat.FormatType, 233
PrintMatrixFormat.ParsePosition, 232
PrintMatrixFormat.RowLabelType, 235
ProblemInfeasibleException, 817
ProblemUnboundedException, 819
QPInfeasibleException, 821
QR, 20
QuadraticProgramming, 140
Quadrature, 76
Quadrature.IFunction, 82
QuasiNewtonTrainer, 711
QuasiNewtonTrainer.IError, 715
RadialBasis, 66
RadialBasis.Gaussian, 71
RadialBasis.HardyMultiquadric, 72
RadialBasis.IFunction, 71
Random, 575
RankException, 922
Ranks, 272
Ranks.Tie, 279
ScaleFactorZeroException, 924
ScaleFilter, 721
ScaleFilter.ScalingMethod, 729
SelectionRegression, 315
SelectionRegression.Criterion, 326
SelectionRegression.SummaryStatistics, 326
Sfun, 171
SignTest, 394
SingularException, 823, 926
SingularMatrixException, 825
Sort, 265
Spline, 37
StepwiseRegression, 327
StepwiseRegression.CoefficientTTestsValue, 335
StepwiseRegression.Direction, 336
Summary, 238
SumOfWeightsNegException, 928
SVD, 23
SymEigen, 31
TableMultiWay, 288
TableMultiWay.TableBalanced, 294
TableMultiWay.TableUnbalanced, 295
TableOneWay, 279
TableTwoWay, 284
TerminationCriteriaNotSatisfiedException, 827
TimeSeriesClassFilter, 741
TimeSeriesFilter, 738
ToleranceTooSmallException, 829
TooManyCallsException, 930
TooManyFunctionEvaluationsException, 932
TooManyIterationsException, 832, 934
TooManyJacobianEvalException, 936
TooManyObsDeletedException, 938
UnboundedBelowException, 834
UnsupervisedNominalFilter, 730
UnsupervisedOrdinalFilter, 733
UnsupervisedOrdinalFilter.TransformMethod, 738
UserBasisRegression, 337
VarBoundsInconsistentException, 836
VarsDeterminedException, 940
Warning, 771
WarningObject, 772
WilcoxonRankSum, 397
WorkingSetSingularException, 838
ZeroFunction, 108
ZeroFunction.IFunction, 111

Index
ZeroNormException, 942
ZeroPolynomial, 104
ZeroSystem, 111
ZeroSystem.IFunction, 113
ZeroSystem.IJacobian, 114