Intel® Integrated Performance Primitives

Reference Manual, Volume 3: Small Matrices and Realistic Rendering

IPP 7.0

Document Number: 319375-020US

Legal Information

Contents

Volume Overview

1

The bulk of this manual describes the structure, operation, and functions of the Intel® Integrated Performance Primitives (Intel® IPP) for small matrices. The manual provides a background for matrix operating concepts used in the Intel IPP software, as well as a detailed description of the respective Intel IPP functions. The Intel IPP functions are combined in groups by their functionality. Each group of functions is described in a separate chapter.

The manual also describes the Intel IPP functions for realistic rendering and 3-dimensional (3D) data processing.

What's New

This Reference Manual documents Intel® Integrated Performance Primitives (Intel® IPP) 7.0 Update 7 release.

Notational Conventions

In this manual, notational conventions include:

- Fonts used for distinction between the text and the code
- Naming conventions for different items.

Font Conventions

The following font conventions are used throughout the manual:

`This type style`	Mixed with the uppercase in structure names as in `IppLibraryVersion`; also used in function names, code examples, and call statements; for example, `ippmAdd_mm_32f`.
`This type style`	Parameters in function type parameters and parameters description, for example, *`src1Stride1`*, *`width`*.

Naming Conventions

The following naming conventions for different items are used by the Intel IPP software:

- All names of the functions used for matrix operations have the `ippm` prefix. In code examples you can distinguish the Intel IPP interface functions from the application functions by this prefix.

NOTE. Each function in this manual is introduced by its short name (without the `ippm` prefix and descriptors) and a brief description of its purpose.

The `ippm` prefix in function names is always used in code examples and function prototypes. In the text, this prefix is omitted when referring to the function group.

- Each new part of a function name starts with an uppercase character without underscore, for example, `ippmFrobNorm`. The underscore is used to separate the data types and data descriptors used with the function, for example, `ippmFrobNorm_ma_32f`.

For the detailed description of the function names structure in Intel IPP, see Function Naming.

Getting Started

2

This chapter explains the purpose and structure of Intel® Integrated Performance Primitives (Intel® IPP) for small matrices.

The chapter also discusses the fundamental concepts used in the small matrix operations part of Intel IPP and defines function naming conventions used in the manual.

Purpose of Intel IPP for Small Matrices

Intel IPP for small matrices (IPP MX) actualizes linear algebra operations on matrices and vectors.

IPP MX provides solutions to a number of software development tasks that include development of various graphics, computer game applications and CAD applications. For example, you can find Intel IPP solutions useful for the applications that require transforming point coordinates from one coordinate system to another, computing dynamics for physical motion modeling, or solving systems of linear equations.

Data Types

Intel IPP for small matrices supports floating-point data with single and double precision.

Memory Layout

IPP MX supports vectors and matrices, whose elements are spaced in memory at equal intervals, as well as matrices and vectors, whose elements are arbitrarily allocated in memory. This feature is unique to this library, more information on the types of matrices and vectors that the functions can operate on is given in IPP MX Objects and Object Description.

Arrays of Vectors and Matrices

Distinctive feature of IPP MX is matrix and vector arrays processing. For example, if there is an IPP MX function that can process two matrices, then there certainly exist analogous functions that process arrays of matrices, with one or both source operands being matrix arrays.

Matrix and vector arrays are processed element by element. Thus, the Add function for vector arrays adds the first vector of the first array to the first vector of the second array, then adds the second vectors of the two arrays, and so forth. The result is stored in the destination vector array. If the first operand is a single vector and the second is a vector array, then the function adds the single vector to each element of the array.

It is recommended to use these functions when processing large amounts of data, as it significantly accelerates your program.

Matrix Transposition

IPP MX library includes special functions that operate on transposed matrices and on arrays of transposed matrices. These functions were provided for such operations as Add, Sub, Mul and some others. For example, there are three IPP MX Add functions that add single matrices: the first function operates on two ordinary matrices, the second function operates on a transposed matrix and an ordinary matrix, and the third one operates on two transposed matrices.

When a function operates on a transposed matrix (or an array of transposed matrices), no special data is required. It is enough to specify non-transposed matrix (array of non-transposed matrices) as function operand, and the function will transpose this matrix (array of matrices) without performance loss.

In-Place Operations

IPP MX library does not contain any in-place operations. Thus, if the source and the destination addresses coincide or overlap, there may be discrepancies in the results shown by versions of Intel IPP optimized for different processors.

Optimization

IPP MX functions are optimized for operations on small matrices and small vectors, particularly for matrices of size 3x3, 4x4, 5x5, 6x6, and for vectors of length 3, 4, 5, 6.

Note that when operating on small arrays, for example, a matrix array made up of two or three matrices of size 3x3, overhead caused by function call and input parameters' check may be greater than the optimization gain.

IPP MX Objects

IPP MX functions operate on the following objects:

- constant
- array of constants
- vector
- array of vectors
- matrix
- array of matrices
- transposed matrix
- array of transposed matrices

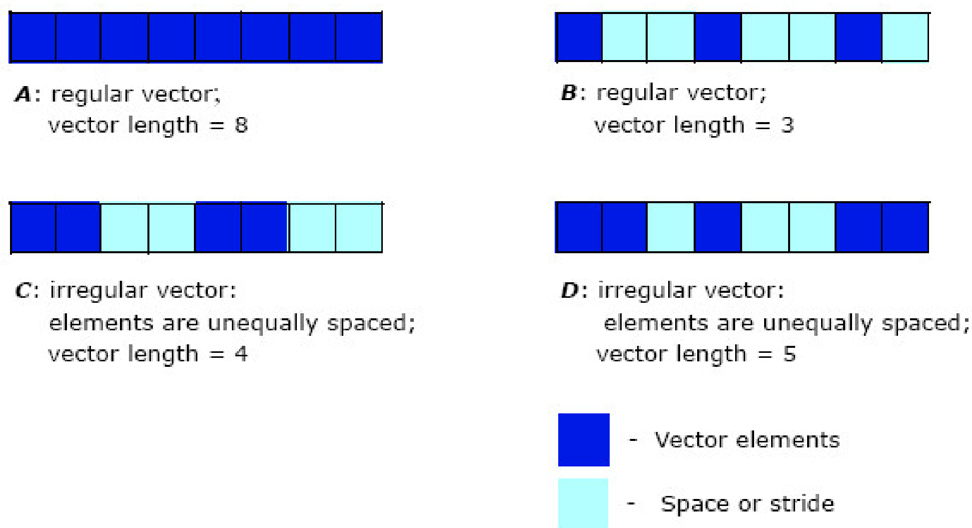
Constant

IPP MX constant is scalar value. Value type is `Ipp32f` or `Ipp64f`.

Vector

The simplest vector is a one-dimensional continuous array (see Figure "Regular and Irregular Vectors", case A). In Intel IPP for small matrices, vectors have a more complicated structure. Any elements stored in memory can be combined into an IPP MX vector. In IPP MX there is a difference between regular and irregular vectors. The vector is called regular, if its elements are equally spaced in memory: see Figure "Regular and Irregular Vectors", cases A, B. Otherwise, the vector is called irregular: see Figure "Regular and Irregular Vectors", cases C, D.

Regular and Irregular Vectors

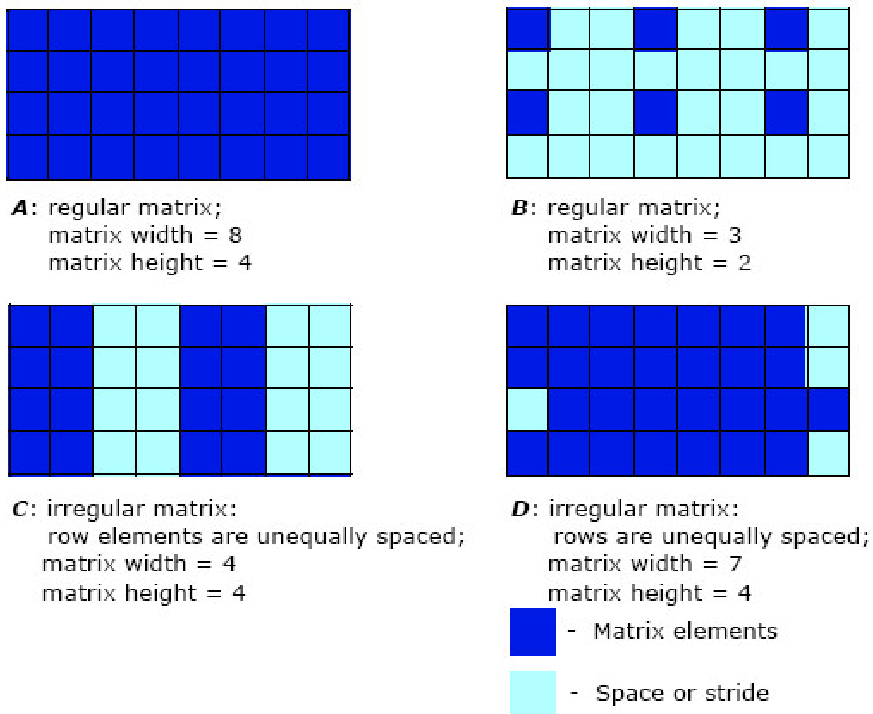


Matrix

The simplest matrix is a two-dimensional continuous array (see Figure "Regular and Irregular Matrices", case A). In Intel IPP for small matrices, matrices are represented by more complicated structures. Any elements stored in memory can be combined into an IPP MX matrix. In IPP MX there is a difference between regular and irregular matrices. The matrix is called regular, if its row elements are equally spaced in memory and its rows are equally spaced in memory: see Figure "Regular and Irregular Matrices", cases A, B. If one or both of these conditions is false, the matrix is called irregular: see Figure "Regular and Irregular Matrices", cases C, D.

In this manual matrix sizes are given as `width x height`.

Regular and Irregular Matrices



Array of Constants

The way Intel IPP for small matrices operates on an array of constants is similar to its operation on a single constant.

All constants in an array are of the same type `Ipp32f` or `Ipp64f`.

Constants in an array may be stored in memory with a regular layout or irregular layout.

An array of constants is stored with a *regular* layout if the constants are equally spaced in memory. Otherwise, the layout is called *irregular*.

The array of constants with a regular layout is defined using an S-pointer (please refer to Figure 2-7).

The array of constants with an irregular layout is defined using an L-pointer (please refer to Figure 2-10).

Array of Vectors

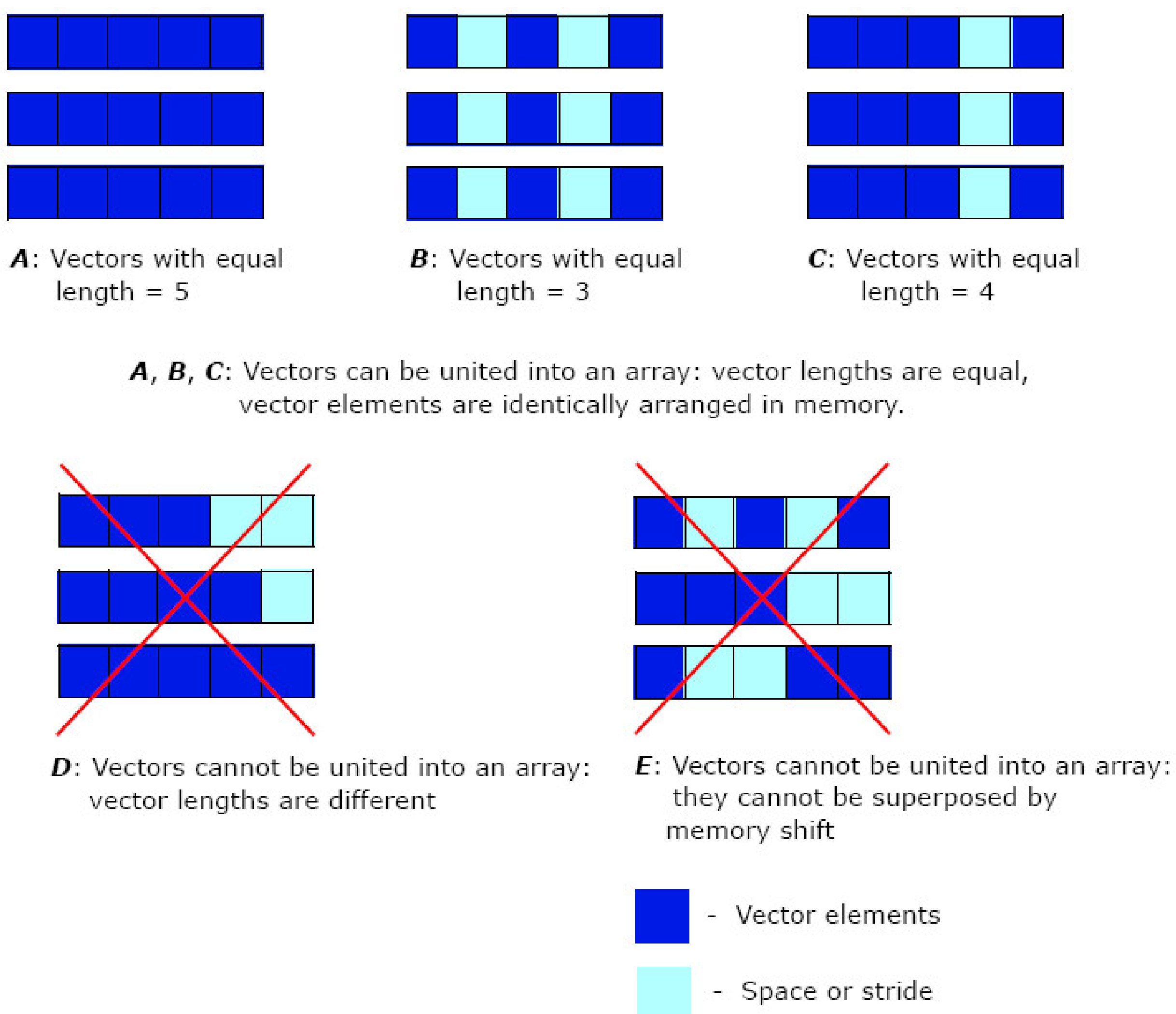
The way Intel IPP for small matrices operates on vector arrays is similar to its operation on single vectors. Thus, different vectors cannot be combined into one array, unless the following conditions are fulfilled:

vectors have equal length

vectors have identical structure, i.e. if superposed by memory shift, they will coincide.

See Figure "Combining Vectors into an Array", cases A, B, C for proper vector sets. If one or both of these conditions is not satisfied, vectors cannot be combined into an array (see Figure "Combining Vectors into an Array", cases D, E).

Combining Vectors into an Array



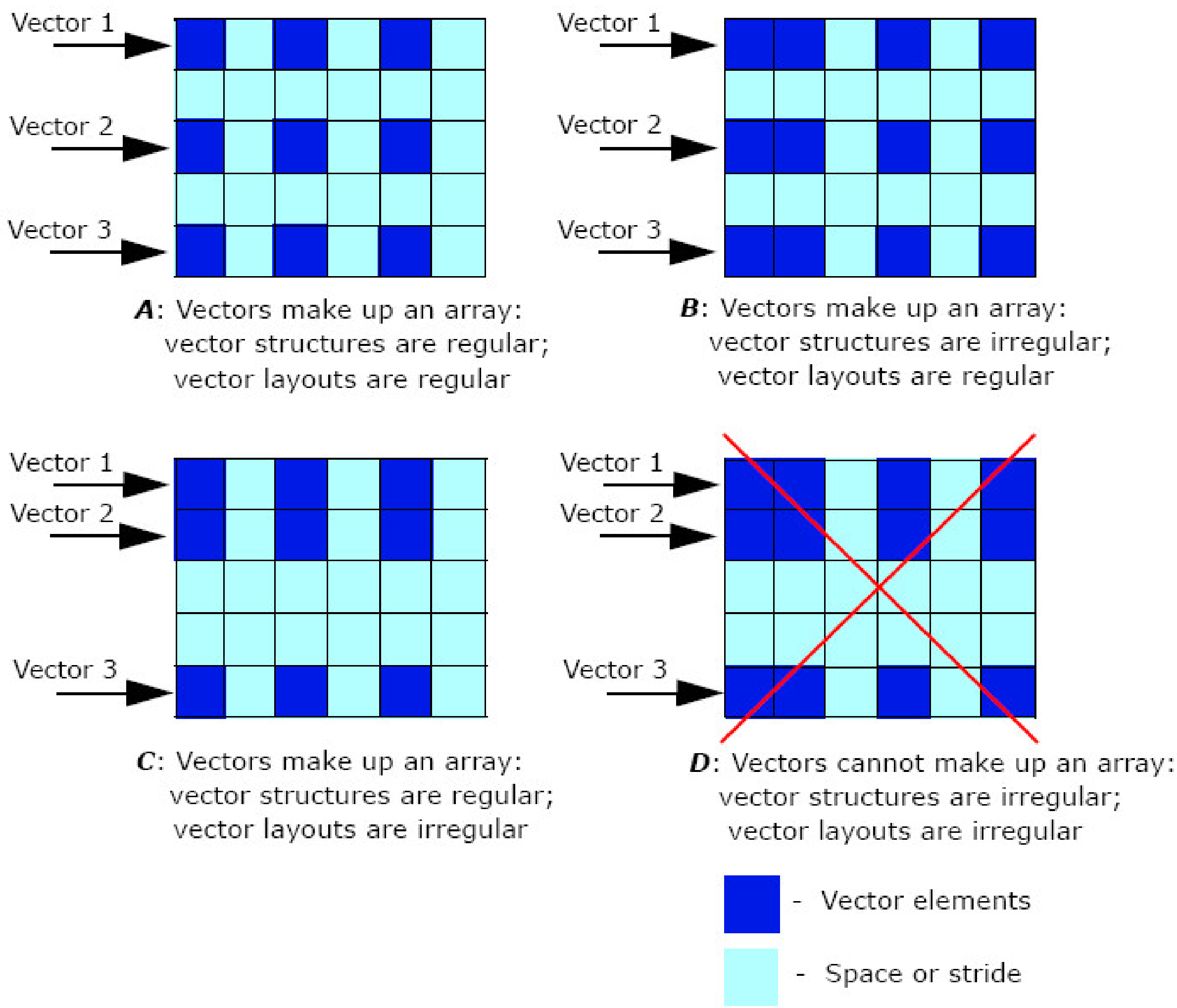
However, not all proper vectors can be combined into an array. Another important condition is vector layout.

The vector layout is called regular, if the vectors are equally spaced in memory; otherwise, the layout is called irregular. In these terms, vectors can be combined in the following way (see Figure "Permissible Vector Arrays", cases A, B, C):

- regular vectors with regular layout
- regular vectors with irregular layout
- irregular vectors with regular layout

Irregular vectors with irregular layout cannot be combined into an array (see Figure “Permissible Vector Arrays”, case D).

Permissible Vector Arrays



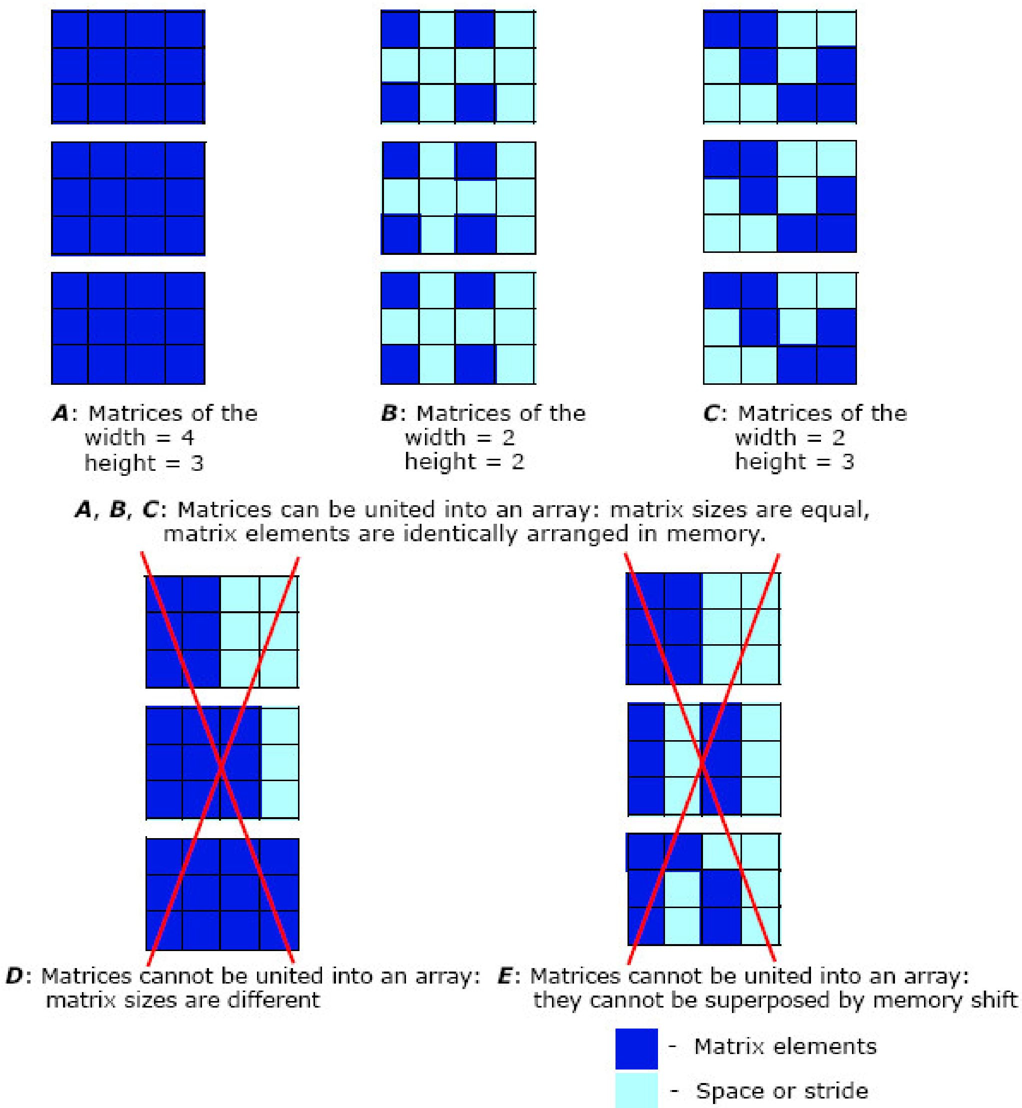
Array of Matrices

The way Intel IPP for small matrices operates on matrix arrays is similar to its operation on single matrices. Thus, different matrices cannot be combined into one array, unless the following conditions are met:

- matrices have equal width and height
- matrices have identical structure, i.e. if superposed by memory shift, they will coincide.

See Figure "Combining Matrices into an Array", cases A, B, C for proper matrix sets. If one or both of these conditions is not met, matrices cannot be combined into an array (see Figure "Combining Matrices into an Array", cases D, E).

Combining Matrices into an Array



However, not all proper matrices can be combined into an array. Another important condition is matrix layout.

The matrix layout is called regular, if the matrices are equally spaced in memory; otherwise, the layout is called irregular. In these terms, matrices can be combined in the following way (see Figure "Permissible Matrix Arrays", cases A, B, C):

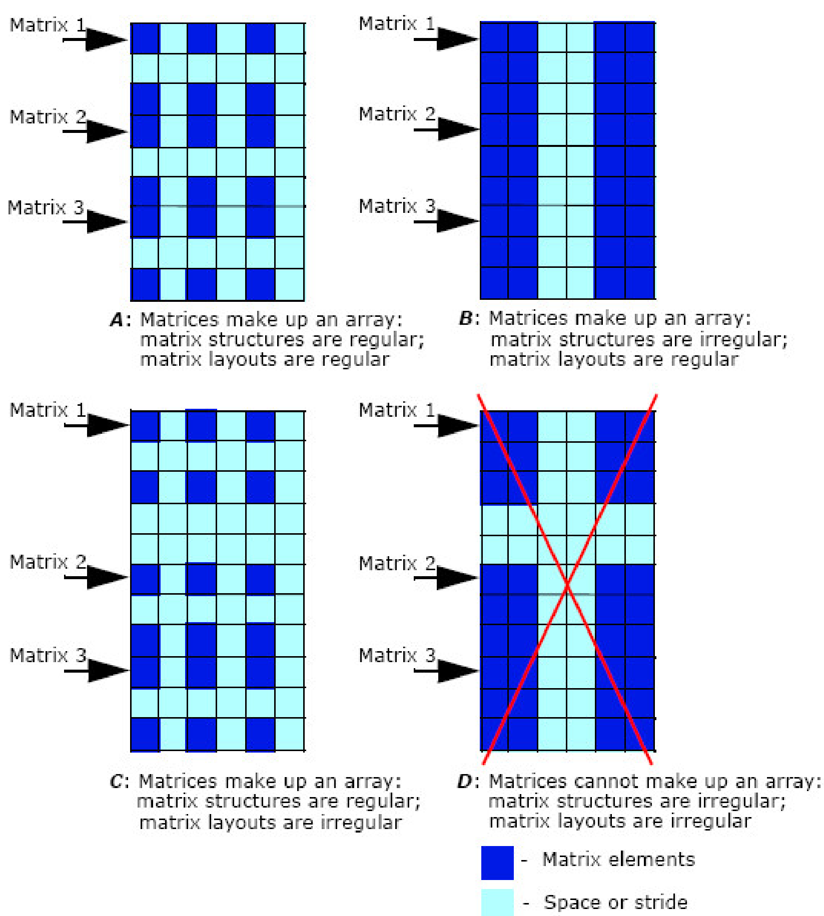
regular matrices with regular layout

regular matrices with irregular layout

irregular matrices with regular layout

Irregular matrices with irregular layout cannot be combined into an array (see Figure "Permissible Matrix Arrays", case D).

Permissible Matrix Arrays



Transposed Matrix

When IPP MX functions operate on transposed matrices, these should be specified as function operands (see Matrix), and the function will transpose the necessary matrix or matrices during calculation.

Array of Transposed Matrices

When IPP MX functions operate on arrays of transposed matrices, the arrays should be specified as function operands (see Array of Matrices), and the function will transpose each matrix in the array during calculation.

Object Description

This section discusses several methods to specify objects that can be used as function arguments. The following terms should be defined before further presentation:

Object	A single matrix, single vector, array of matrices, array of vectors or array of constants that can serve as IPP MX function operand.
Object size	Object data that will be used by the IPP MX function for calculation, i.e. width and height of a matrix and length of a vector. If the operation is performed on a matrix or a vector array, object size is equal to the size of a single element in the array.

Description Methods

IPP MX library provides two methods of the description of a single matrix or a vector:

Standard description	The method is used when the matrix (vector) is regular (Figure "Regular and Irregular Vectors" and Figure "Regular and Irregular Matrices", cases A and B).
Pointer description	The method is used when the matrix (vector) is irregular (Figure "Regular and Irregular Vectors" and Figure "Regular and Irregular Matrices", cases C and D).

Note that objects describable using the Standard method can be also represented through the Pointer description but not vice versa.

IPP MX library provides three description methods for arrays of matrices, vectors and constants:

Standard description	The method is used when all the matrices (vectors) have regular structure. Matrices (vectors, constants) must be regularly spaced in memory (Figure "Permissible Vector Arrays" and Figure "Permissible Matrix Arrays", case A).
Pointer description	The method is used when all the matrices (vectors) have irregular structure. Matrices (vectors) must be regularly spaced in memory (Figure "Permissible Vector Arrays" and Figure "Permissible Matrix Arrays", case B).
Layout description	The method is used when all the matrices (vectors) have regular structure, but are irregularly spaced in memory (Figure "Permissible Vector Arrays" and Figure "Permissible Matrix Arrays", case C) or when the constants are irregularly spaced in memory.

Note that arrays of objects describable using the Standard method can be also represented through the Pointer or Layout description but not vice versa.

NOTE. All elements in the array must have identical structure.

The following subsections describe the above methods in detail by means of IPP MX function parameters. There is no specification of object size, since although the size is an essential feature of an object, many IPP MX functions require only one size for several objects. The concept of size as a function and object attribute, as well as its specification is described in Object Size Puzzle.

Strides

If the data is regularly organized, it is easier to describe it in terms of strides. Intel IPP for small matrices introduces three types of strides:

Stride 0	stride between matrices, vectors, or constants in the array.
Stride 1	stride between matrix rows.
Stride 2	stride between vector elements or matrix row elements.

When operating on regular matrices, you should specify stride1 and stride2.

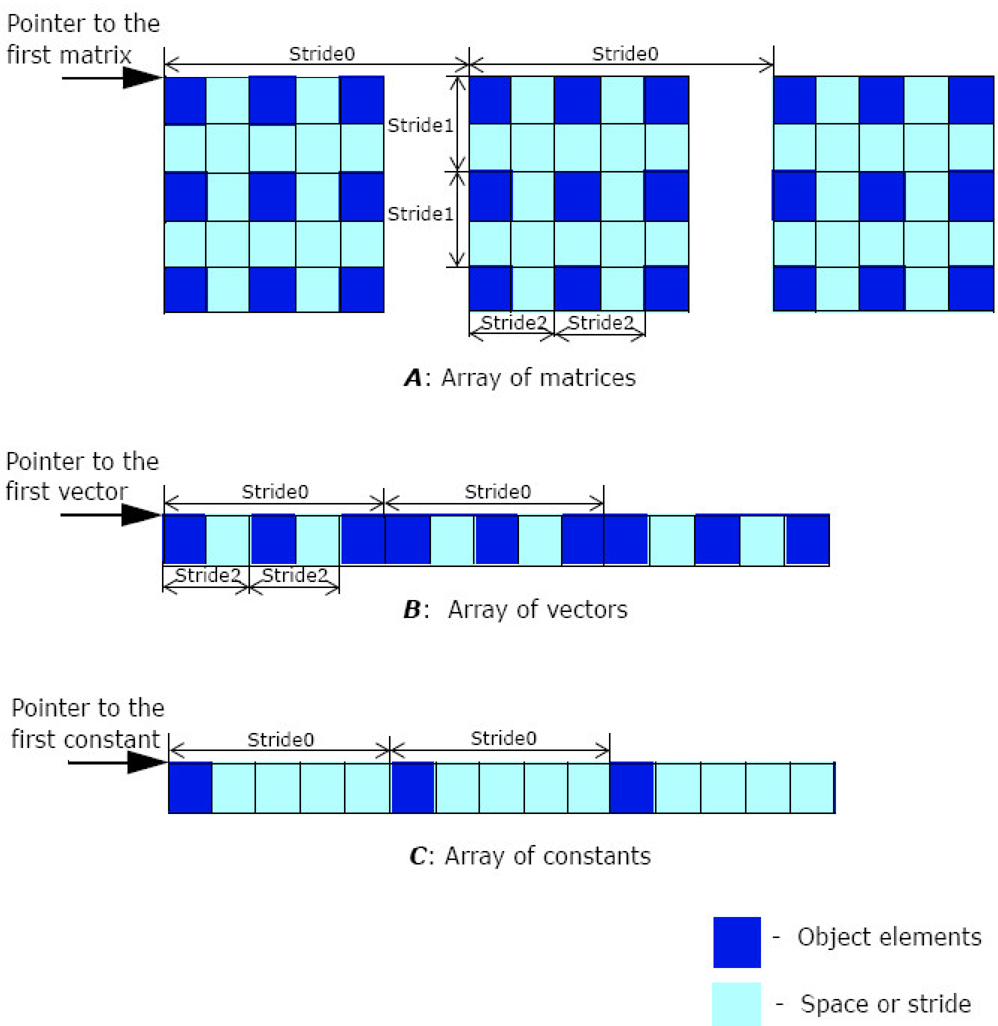
When operating on transposed matrices, you should never exchange stride1 and stride2. Instead, use the function that operates on transposed matrices.

NOTE. All strides are measured in bytes. Stride value must be positive and divisible by the size of the data type. To convert stride value measured in elements to the number of bytes you should multiply it by the size of the data type.

Standard Description

To describe an object by Standard method, specify one pointer to object's data and several strides. When the operation is performed on a single matrix or a vector, the required pointer is the pointer to the first object element. When the operation is performed on matrix or vector arrays, the required pointer is the pointer to the first element in the first matrix (vector) of an array. When the operation is performed on arrays of constants, the required pointer is the pointer to the first constant of an array.

Standard Description



The following fragment of C code describes a regular matrix array shown in Figure "Standard Description", A:

```
// Allocate memory for the array
of continuous matrices

Ipp32f pMatrices[5*5*3];

// Set stride2

int stride2 = sizeof(Ipp32f)*2;

// Set stride1

int stride1 = sizeof(Ipp32f)*10;

// Set stride0

int stride0 = sizeof(Ipp32f)*25;

   * * *

// Call IPP MX function

ippm<Operation>_ma_32f(..., pMatrices, stride0, stride1, stride2, ...);
```

The following code fragment represents description for a regular array of vectors shown in Figure "Standard Description", B:

```
// Allocate memory for the array
of continuous vectors

Ipp32f pVectors[5*3];

// Set stride2

int stride2 = sizeof(Ipp32f)*2;

// Set stride0

int stride0 = sizeof(Ipp32f)*5;

   * * *

// Call IPP MX function

ippm<Operation>_va_32f(...,pVectors, stride0, stride2, ...);
```

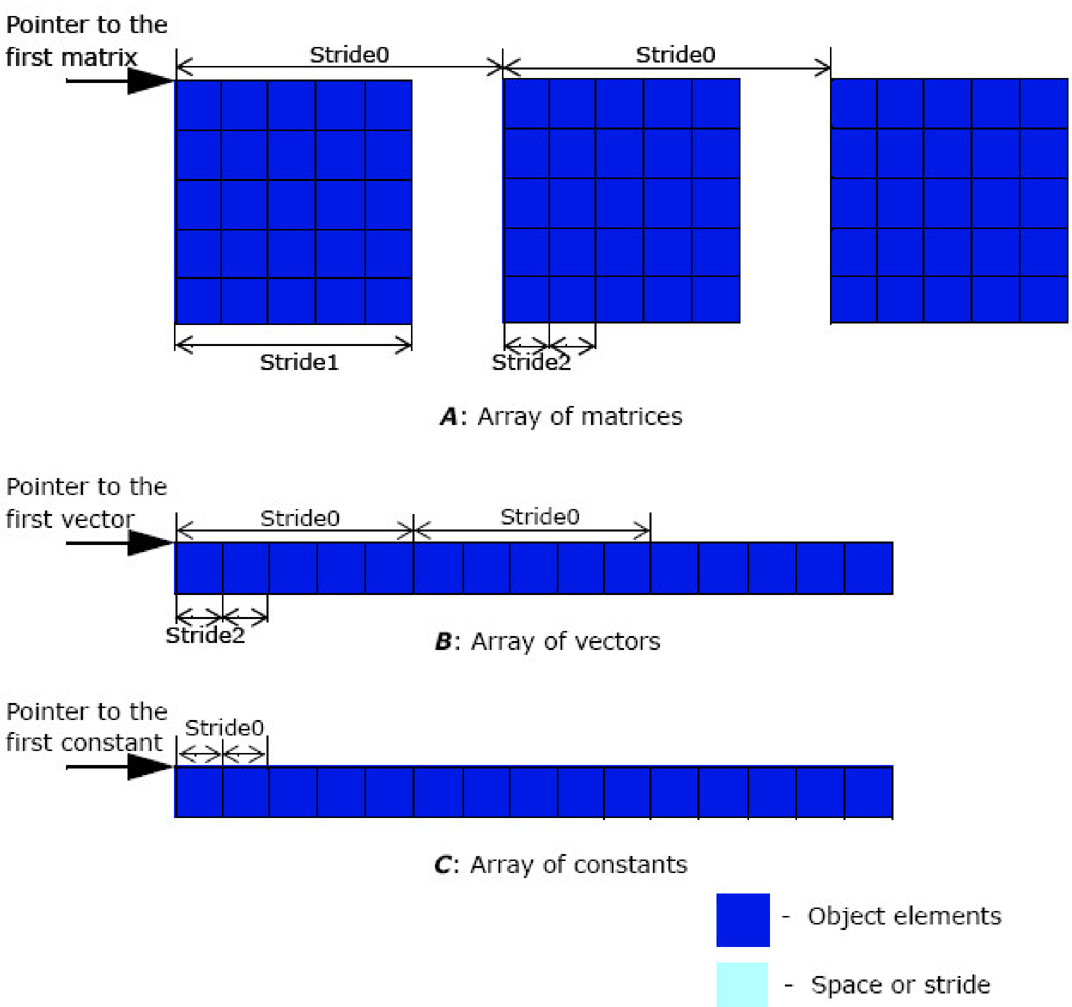
Single matrices and vectors are described in the same way without the stride 0 specification.

The following code fragment represents description for a regular array of constants shown in Figure "Standard Description", C:

```
// Allocate memory for the array
of continuous constants

// example is for count=3

Ipp32f pVal[5*3];

// Set stride0

int valStride0 = sizeof(Ipp32f)*5;

* * *

// Call IPP MX function

ippm<Operation>_ca_32f(..., pVal, valStride0, ...);
```

NOTE. Strides are calculated in bytes.

Standard Description (Continuous Object)



Continuous objects also belong to the general regular case. Thus, all strides including stride 2, must be specified.

When operating on a matrix array, shown in Figure "Standard Description (Continuous Object)", case A, specify the following strides (strides are calculated for Ipp32f data type):

```
int stride2 = sizeof(Ipp32f);

int stride1 = stride2*width;

int stride0 = stride1*height;
```

When operating on a vector array, shown in Figure “Standard Description (Continuous Object)”, case B, specify the following strides

```
int stride2 = sizeof(Ipp32f);

int stride0 = stride2*length;
```

When operating on an array of constants, shown in Figure “Standard Description (Continuous Object)”, case C, specify the following stride:

```
int stride0 = sizeof(Ipp32f);
```

Pointer Description

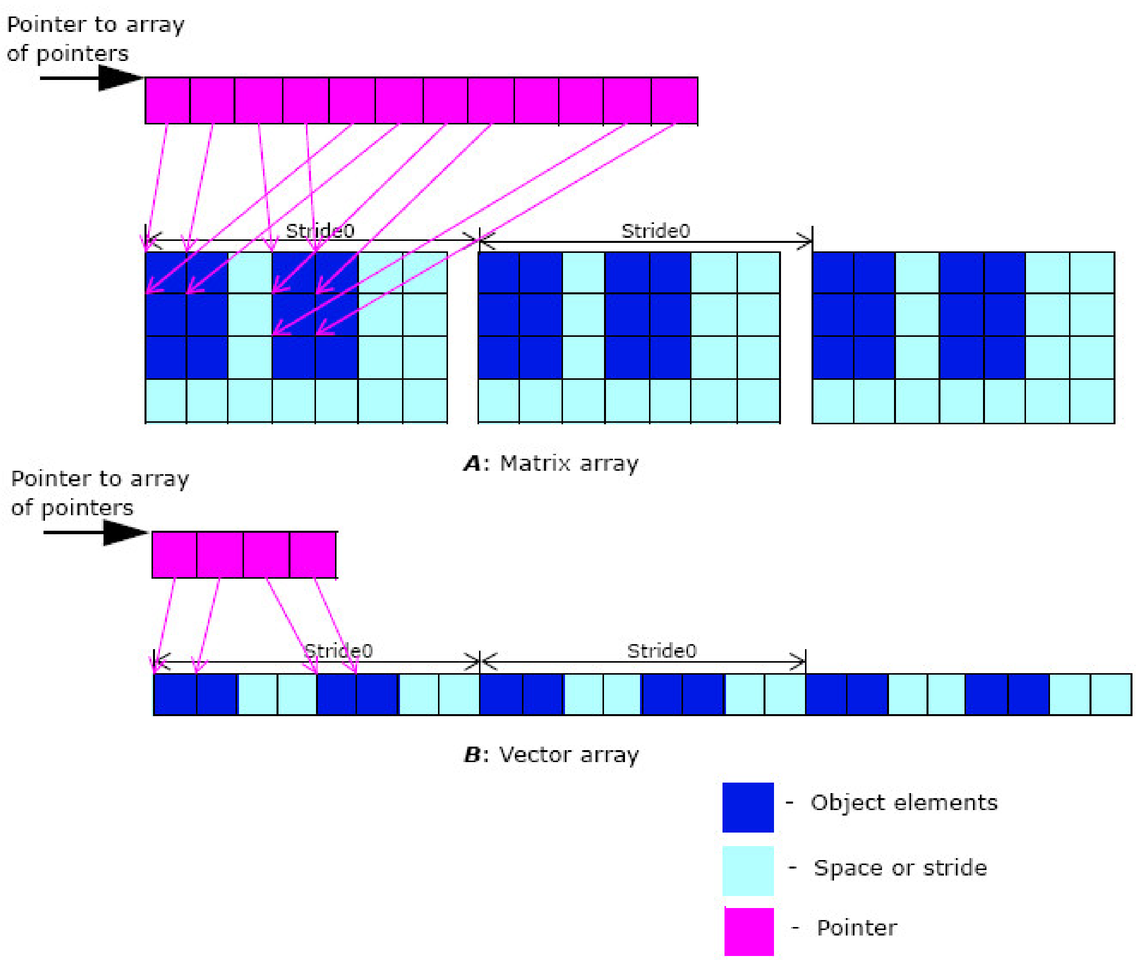
Pointer method is used when you deal with objects of irregular structure. A convenient way to describe an irregular structure is creation of a mask for the data. The Pointer method creates masks for objects by direct pointing to every element of a vector (matrix). Pointers to the elements make up an array.

To describe an object by the Pointer method, specify the pointer to the array of pointers and roiShift in bytes (The concept of the roiShift parameter will be clarified later; in this discussion it is set to 0.) If the operation is performed on matrix or vector arrays, also specify the stride between the matrices (vectors), i.e. stride 0.

To apply the Pointer description method, first, create an array of pointers with the length equal to the number of elements in a single matrix (vector). Then initialize this array by making its elements point to every element of the single matrix (vector). When processing matrix or vector arrays, the pointers must point to the elements of the first matrix (vector).

IPP MX function that operates on objects described by the Pointer method has suffix _P. If a function has the _P suffix, then all objects, except constants, must be defined by Pointer method. Array of constants must be described by the Standard method. A singe constant is described as a value.

Pointer Description



The following fragment of C code describes the matrix array shown in Figure “Pointer Description”, case A:

```
// Allocate memory for the array of continuous matrices
Ipp32f  pMatrices[8*4*3];
// Assign pointer to the first continuous matrix
Ipp32f* pFirstMatrix = pMatrices;
// Allocate special array for pointers to matrix elements
Ipp32f* ppPointer[4*3];
// Declare subsidiary descriptors
int roiShift;
int stride0;
   * * *
// Set first row of matrix
ppPointer [0] = pFirstMatrix + 0;
ppPointer [1] = pFirstMatrix + 1;
ppPointer [2] = pFirstMatrix + 4;
ppPointer [3] = pFirstMatrix + 5;
// Set second row of matrix
ppPointer [4] = pFirstMatrix + 8 + 0;
ppPointer [5] = pFirstMatrix + 8 + 1;
ppPointer [6] = pFirstMatrix + 8 + 4;
ppPointer [7] = pFirstMatrix + 8 + 5;
// Set third row of matrix
ppPointer [ 8] = pFirstMatrix + 8*2 + 0;
ppPointer [ 9] = pFirstMatrix + 8*2 + 1;
ppPointer [10] = pFirstMatrix + 8*2 + 4;
ppPointer [11] = pFirstMatrix + 8*2 + 5;
// Set roiShift
roiShift = 0;
// Set stride0
stride0 = sizeof(Ipp32f)*8*4;
// Call IPP MX function
ippm<Operation>_ma_32f_P(..., ppPointer, roiShift, stride0, ...);
```

The following fragment of C code describes the vector array shown in Figure “Pointer Description”, case B:

```
// Allocate memory for the array of continuous vectors
Ipp32f  pVectors[8*3];
// Assign pointer to the first continuous vector
Ipp32f* pFirstVector = pVectors;
// Allocate special array for pointers to vector elements
Ipp32f* ppPointer[4];
// Declare subsidiary descriptors
int roiShift;
int stride0;
   * * *
// Set pointers
ppPointer [0] = pFirstVector + 0;
ppPointer [1] = pFirstVector + 1;
ppPointer [2] = pFirstVector + 4;
ppPointer [3] = pFirstVector + 5;
// Set roiShift
roiShift = 0;
// Set stride0
stride0 = sizeof(Ipp32f)*8;
// Call IPP MX function
ippm<Operation>_va_32f_P(..., ppPointer, roiShift, stride0, ...);
```

Single matrices and vectors are described in the same way without stride 0 specification.

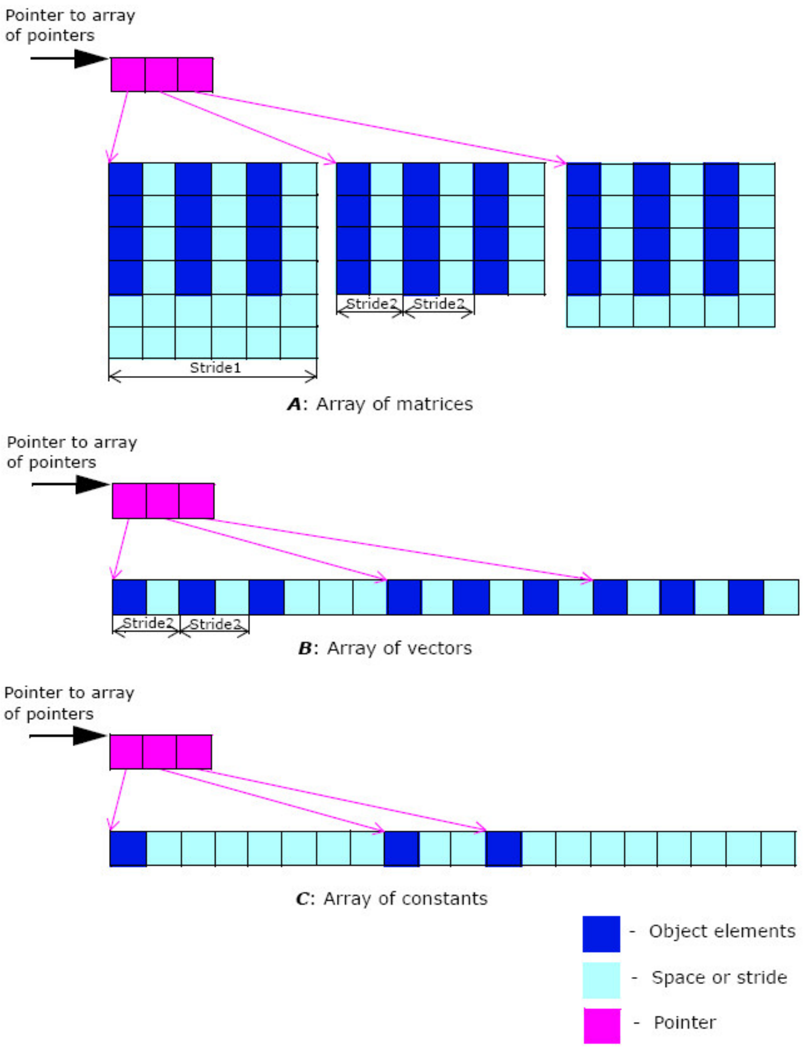
Layout Description

Layout description method is used when dealing with arrays of matrices, vectors or constants. Unlike the Pointer description method, which defines matrix elements by pointers and matrix layout by strides, the Layout method defines matrix elements by strides and matrix layout by pointers.

To describe an object by the Layout method, specify pointers to each matrix, vector or constant in the array, roiShift (in bytes), stride 2, and stride 1 (The concept of the roiShift parameter will be clarified later; in this discussion it is set to 0.). Note that no stride 0 specification is required. Create a special array of pointers with the length equal to the number of array components. Initialize the array by making its elements point to every matrix (vector or constant). Specify the strides required to define single matrix (vector).

IPP MX function that operates on objects described by the Layout method has suffix _L. If a function has the _L suffix, then arrays of matrices, vectors or constants must be defined by the Layout method and single matrices or vectors by the Standard method.

Layout Description



The following fragment of C code describes the matrix array shown in Figure “Layout Description”, case A:

```
// Allocate memory for three continuous matrices
Ipp32f pFirstMatrix [6*5];
Ipp32f pSecondMatrix[6*5];
Ipp32f pThirdMatrix [6*5];
// Allocate special array for pointers to 3 matrices:
Ipp32f* ppLayout[3];
// Declare subsidiary descriptors
int roiShift;
int stride1, stride2;
   * * *
// Set pointers to matrices
ppLayout [0] = pFirstMatrix;
ppLayout [1] = pSecondMatrix;
ppLayout [2] = pThirdMatrix;
// Set roiShift
roiShift = 0;
// Set stride2
stride2 = sizeof(Ipp32f)*2;
// Set stride1
stride1 = sizeof(Ipp32f)*6;
// Call IPP MX function
ippm<Operation>_ma_32f_L(..., ppLayout, roiShift, stride1, stride2, ...);
```

The following fragment of C code describes the vector array shown in Figure "Layout Description", case B:

```
// Allocate memory for three continuous vectors
Ipp32f pFirstVector [6];
Ipp32f pSecondVector[6];
Ipp32f pThirdVector [6];
// Allocate special array for pointers to 3 vectors
Ipp32f* ppLayout[3];
// Declare subsidiary descriptors
int roiShift;
int stride2;
   * * *
// Set pointers to vectors
ppLayout [0] = pFirstVector;
ppLayout [1] = pSecondVector;
ppLayout [2] = pThirdVector;
// Set roiShift
roiShift = 0;
// Set stride2
stride2 = sizeof(Ipp32f)*2;
// Call IPP MX function
ippm<Operation>_va_32f_L(..., ppLayout, roiShift, stride2, ...);
```

The following fragment of C code describes the constant array shown in Figure "Layout Description", case C:

```
// Allocate memory for constant array Ipp32f pVal [20];

// Allocate special array for pointers to 3 constants

// example is for count=3

Ipp32f* ppValLayout[3];

// Declare subsidiary descriptors

int valRoiShift;

* * *

// Set pointers to constants

ppValLayout [0] = &pVal[0];

ppValLayout [1] = &pVal[8];

ppValLayout [2] = &pVal[11];

// Set roiShift

valRoiShift = 0;

* * *

// Call IPP MX function

ippm<Operation>_ca_32f_L(..., ppValLayout, valRoiShift, ...);
```

RoiShift Parameter

To illustrate the concept of the roiShift (region of interest shift) parameter, have another look at Figure "Layout Description". In the described objects, odd columns of continuous matrices and odd elements of continuous vectors are processed. However, to process even columns or elements, an additional parameter may be needed to specify the shift (in this case, one-element shift right).

If the Standard description method is used, to describe data shifted this way, it is sufficient to shift only one data pointer. However, with the Layout method used, it is necessary to shift every pointer in the layout array by the same number of elements (in this case, one), or bytes. Similar problem arises when the Pointer description is used: to shift the mask, all elements in the pointer array must be shifted accordingly. This shift is specified using the roiShift parameter, which is required for Pointer and Layout description methods. RoiShift specifies the number of bytes by which IPP MX function will shift each pointer in the pointer array. If no shifting is needed (as in Figure "Pointer Description" and Figure "Layout Description"), roiShift is set to 0.

NOTE. RoiShift parameter is measured in bytes. RoiShift value can be equal to 0. Otherwise, roiShift value must be positive and divisible by the size of the data type. To convert roiShift value measured in elements to the number of bytes you should just multiply it by the size of the data type.

Pointer Description with RoiShift

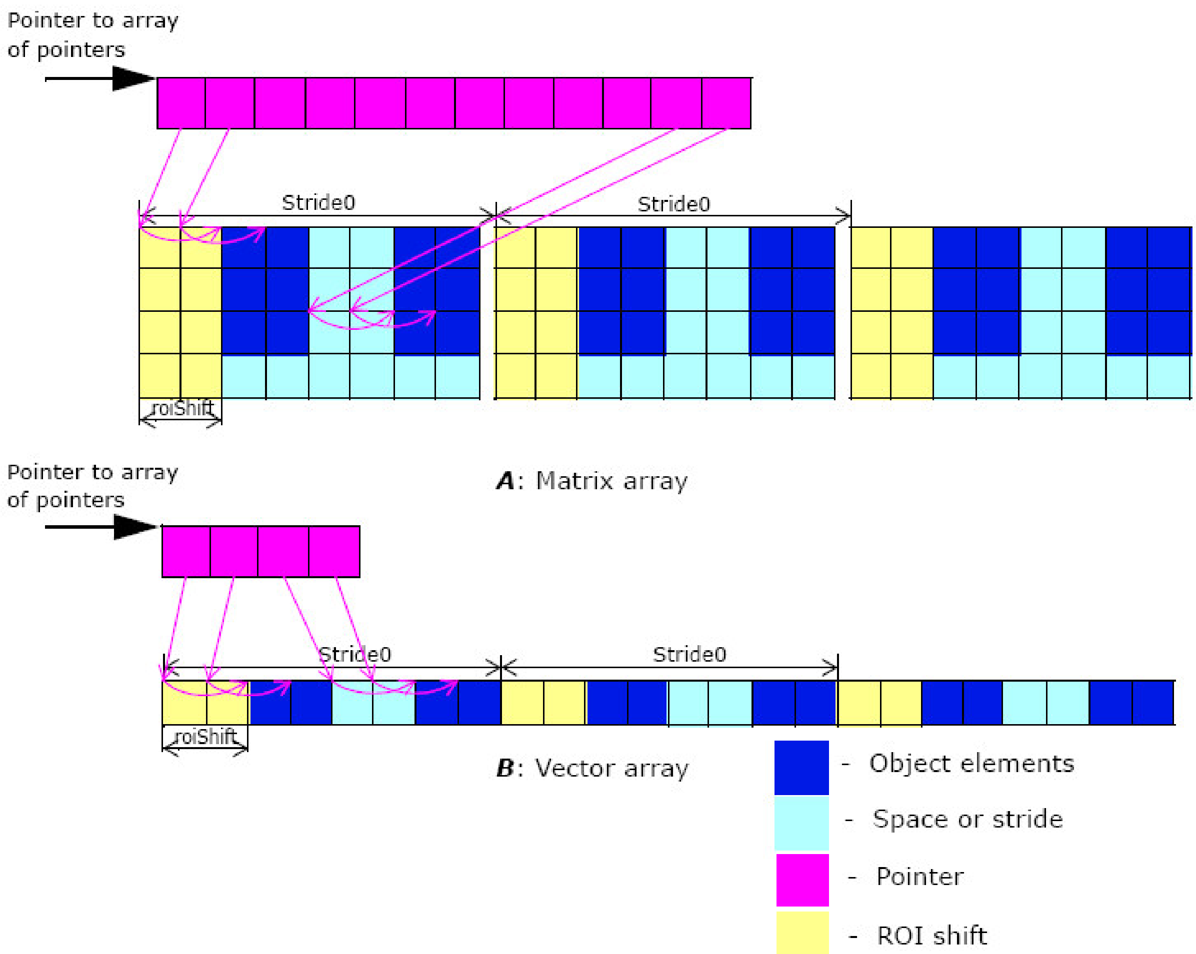


Figure "Pointer Description with RoiShift", case A shows the matrix array that is a result of Pointer description with non-zero roiShift parameter.

Figure "Pointer Description with RoiShift", case B shows the vector array that is a result of Pointer description with non-zero roiShift parameter.

The following fragment of C code specifies roiShift for both A and B cases:

```
// Set roiShift

roiShift = 2*sizeof(32f);
```

Object Descriptors Table

The following table contains subsidiary object descriptors required for each particular case. All description methods and all object types are collected here. The order of descriptors in the table is the order in which IPP MX function parameters are presented.

Object Descriptors

Description	Object	roiShift	stride0	stride1	stride2
Standard	Matrix	-	-	+	+
	Array of matrices	-	+	+	+
	Vector	-	-	-	+
	Array of vectors	-	+	-	+
	Array of constants	-	+	-	-
Pointer	Matrix	+	-	-	-
	Array of matrices	+	+	-	-
	Vector	+	-	-	-
	Array of vectors	+	+	-	-
Layout	Array of matrices	+	-	+	+
	Array of vectors	+	-	-	+
	Array of constants	+	-	-	-

Function Naming

The function names in the small matrices domain of Intel IPP begin with the `ippm` prefix and have the following general format:

```
ippm<name>_<objects>_<datatype>[_<descriptor>](<arguments>);
```

Name

The `<name>` is an abbreviation for the core function operation, for example, “`Add`”, “`Copy`”. The `<name>` may consist of several functional parts. Each new part of a function name starts with an uppercase character, without underscore, for example, `ippmFrobNorm`.

Objects

```
objects = <objecttype1>[<objecttype2>][<objecttype3>]
```

Object type describes the type of source objects passed to a function for processing and may be the following:

`c`	constant
`ca`	array of constants
`v`	vector
`va`	array of vectors
`m`	matrix
`t`	transposed matrix
`ma`	array of matrices

`ta`	array of transposed matrices.

If the function has only one source object, then `objects = <objecttype1>`.

In case of two source objects, `objects = <objecttype1><objecttype2>`. Even if both source objects have the same type, double type is required.

The one or two object types in the function name that are *required* for a particular function determine which algebraic operation the function carries out.

For example,

`Mul_vc` - multiplication of a vector by a constant,

`Mul_mc` - multiplication of a matrix by a constant,

`Mul_mv` - multiplication of a matrix by a vector,

`Mul_mm` - multiplication of a matrix by a matrix.

Unlike functions operating on two source objects, for functions with three source objects, if the second and third objects have the same type, then, *by default,* this type is specified only once:

`objects = <objecttype1><objecttype2>`,

where `<objecttype2>` is the type of the second and the third object. For example, the function name `Gaxpy_mva` indicates that both the second and the third source objects are arrays of vectors.

If the second and third source objects have different types, all the three types are required to be specified: `objects = <objecttype1><objecttype2><objecttype3>`. For example, for `Gaxpy_mvav`, the second source object is an array of vectors and the third source object is a single vector.

The type of destination object is declared, if the function has no source objects. Otherwise, the destination type is determined by types of source objects and the operation the function carries out.

Data Types

The current version of Intel IPP supports the following data types of the source and destination for functions that perform matrix operations:

`32f`	32-bit floating-point
`64f`	64-bit double precision.

All objects of each particular IPP MX function must have the same type. Accordingly, it is assumed that sizes of the objects are specified in sizes of the data type, unless otherwise explicitly indicated.

Descriptor

The `<descriptor>` field defines object description method (see Object Description) and contains the following symbols:

`S`	Standard description.
`P`	Pointer description.
`L`	Layout description.

The default for Intel IPP matrix operating functions is the Standard description method.

Practically all IPP MX function names have one descriptor symbol or no descriptor at all, which allows them to be specified by Standard object description. The only exception is Copy functions that can copy data from one description to another. Copy function name contains two descriptors to define the source and the destination description methods.

If IPP MX function name has the L descriptor, while the function itself operates not only on matrix and vector arrays, but also on single matrices (vectors), then the single matrix or vector must be described by Standard method, since the Layout method does not work on single objects.

Arguments

The `<arguments>` field specifies the function parameters. The order of arguments is as follows:

- all source objects (constants follow matrices and vectors)
- all destination objects
- count - number of matrices (vectors, constants) in arrays (if the function operates on arrays of the respective objects)
- other, operation-specific operands.

The order of arguments specifying non-constant object is as follows:

- pointer to object data
- subsidiary object descriptors (see Object Descriptors Table)
- object size - optional arguments.

Object size is not obligatory for all objects. Object size arguments may be as follows:

- *width*, *height* - matrix width and height
- *widthHeight* - square matrix width and height
- *length* - vector length.

Parameter Name Convention

The parameter name has the following conventions:

- All parameters defined as pointers to any object start with p, for example, *pSrc*, *pDst*; all parameters defined as double pointers (pointers to the pointers) start with *pp*, for example, *ppSrc*, *ppDst*.
- All parameters defined as values start with a lowercase letter, for example, *val*, *len*, *count*.
- Each new part of a parameter name starts with an uppercase letter, without underscore, for example, *pSrc*, *srcStride2*.
- Each parameter name specifies its functionality. Source parameters named *pSrc* or *src* are sometimes followed by names or numbers, for example, *pSrc2*, *src2Len*.
- Output parameters named pDst or dst are followed by names or numbers, for example, *pDst*, *dstLen*.

Operations with Arrays of Objects

Processing of matrix and vector arrays is a distinctive feature of IPP MX. Each Linear Algebra operation is implemented in the IPP MX interface with a base function, operating on single objects (matrices, vectors, or constants), and a collection of functions carrying out the same operation with one, two, or all source operands being arrays of matrices, vectors, or constants.

IPP MX base functions for matrices and vectors of certain small sizes (particularly, matrices of size 3x3, 4x4, 5x5, 6x6 and vectors of length 3, 4, 5, 6) are rather of supplementary importance, as their performance may sometimes be even worse than of a direct C code. The IPP MX interface mainly optimizes the use of functions implementing Linear Algebra operations with *arrays* of matrices, vectors and constants.

This section explains which operands composing source arrays of matrices, vectors, or constants such functions actually operate on and how the result is composed. Types of function operands are reflected in the function names (see the Function Naming section).

The discussion uses the following notation:

<ops>	An algebraic operation on matrices, vectors, and constants.
src, src1, src2	A single, first or second source vector, respectively.
dst	A destination vector.
src [*i*]	*i*-th element of the source vector.
dst [*i*]	*i*-th element of the destination vector.
src_j , $src1_j$, $src2_j$	*j*-th vector (matrix) in a single, first or second source array of vectors (matrices), respectively.
dst_j	*j*-th vector (matrix) in a destination array of vectors (matrices), respectively.
val	A constant.
Val_j	*j*-th constant in a source array of constants.
len	Vector length.
count	The number of matrices, vectors or constants in each array.

Vector-constant and Matrix-constant Operations

This subsection expands on vector-constant operations with arrays of vectors and constants. The matrix-constant operations with arrays of matrices and constants are carried out in a similar way.

The base function (object type is "vc" or "cv") performs an operation with a constant and a vector so that the operation is carried out with each element of the source vector and the constant. The result is stored in the destination vector:

$dst[i] = src[i] <ops> val$, $0 \le i < len$.

When the first operand is a vector array and the second operand is a single constant (object type is "vac" or "cva"), the operation is carried out with the constant and each element of the j-th source vector. The result is stored in j-th destination vector:

$dst_j[i] = src_j[i] <ops> val$, where $0 \le i < len$, $0 \le j < count$.

When the first operand is a vector array and the second operand is an array of constants (object type is "vaca" or "cava"), the operation is carried out with each element of the j-th source vector and the j-th constant in the source array of constants `Val`. The result is stored in j-th destination vector:

$dst_j[i] = src_j[i] <ops> Val_j$, where $0 \le i < len$, $0 \le j < count$.

When the first operand is a single vector and the second operand is an array of constants (object type is "vca" or "cav"), the operation is carried out with each element of the source vector and the j-th constant in the source array of constants `Val`. The result is stored in j-th destination vector:

$dst_j[i] = src[i] <ops> Val_j$, where $0 \le i < len$, $0 \le j < count$.

For example, see the Add function in chapter 4.

Operations with One Vector/Matrix Array

The base function carries out an operation with one source vector or matrix (object type is "v" or "m"):

dst = <*ops*> *src*, which means that an appropriate operation is carried out with vector or matrix elements.

The respective vector (matrix) array function (object type is "va" or "ma") applies the base function to each vector (matrix) in the source array and stores the result in the destination array:

dst_j = <*ops*> src_j , $0 \le j < count$.

For example, see the Copy function in chapter 3.

Operations with Matrix and Vector Arrays

This subsection expands upon vector-vector, matrix-vector and matrix-matrix *binary* operations with arrays of matrices and vectors. Matrix (vector) array functions having three or more source objects operate in a similar way, as well as functions with transposed matrices.

The base function operates on two sources: two vector operands, two matrix operands or a matrix and vector operand (object type is "vv", "mm", or "mv"):

dst = *src1* <*ops*> *src2*, which means that an appropriate operation is carried out with vector or matrix elements.

When the first source operand of the function is an array of matrices (vectors) and the second operand is a single matrix (vector) (object type is "vav", "mav", or "mam"), the base function is applied to each *j*-th matrix (vector) in the source array (first operand) and the single source matrix or vector (second operand). The result is stored in the *j*-th destination matrix (vector).

dst_j = $src1_j$ <*ops*> *src2*, $0 \le j < count$.

When the first source operand of the function is a single matrix (vector) and the second operand is an array of matrices (vectors) (object type is "vva", "mva", or "mma"), the base function is applied to the single source matrix or vector (first operand) and each *j*-th matrix (vector) in the source array (second operand). The result is stored the *j*-th destination matrix (vector).

dst_j = *src1* <*ops*> $src2_j$, $0 \le j < count$.

When both source operands of the function are arrays of matrices (vectors) (object type is "vava", "mava", or "mama"), the base function is applied to each *j*-th matrix (vector) in the first source array and each *j*-th source matrix (vector) in the second source array. The result is stored in the *j*-th destination matrix (vector).

*dst*j = $src1_j$ <*ops*> $src2_j$, $0 \le j < count$.

For example, see the Saxpy function in chapter 4 and the Mul function in chapter 5.

Object Size Puzzle

As it was said in the Function Naming section (see Arguments), object size is not always required when describing an object. The majority of IPP MX functions specify several objects and only one object size. There exist certain rules that define the object size parameter in such cases:

- If the object is followed by object size, this is the size of the object.

- If there is no object size, this parameter is calculated by another object's size based on the function purpose and object types.

Example 1:

```
IppStatus ippmTranspose_m_32f(const Ipp32f* pSrc, int srcStride1,
    int srcStride2, int width, int height, Ipp32f* pDst, int dstStride1,
    int dstStride2);
```

Object size `width`, `height` follows the `src` matrix, therefore, it is the `src` size.

srcWidth = width, *srcHeight = height*

The function purpose is transposition, therefore

dstWidth = height, *dstHeight = width*

Example 2:

```
IppStatus ippmAdd_mm_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2,
    Ipp32f* pDst, int dstStride1, int dstStride2, int width, int height);
```

Object size `width`, `height` follows the `dst` matrix, therefore, it is the `dst` size.

dstWidth = width, *dstHeight = height*

The function purpose is addition of matrices, therefore the sizes of the elements should be equal to the size of `dst`:

src1Width = width, *src1Height = height*

src2Width = width, *src2Height = height*

Example 3:

```
IppStatus ippmAdd_tm_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f* pDst,
    int dstStride1, int dstStride2, int width, int height);
```

Object size `width`, `height` follows the `dst` matrix, therefore, it is the `dst` size.

dstWidth = width, *dstHeight = height*

The function purpose is addition of matrices, therefore the sizes of the elements should be equal to the size of `dst`.

item1Width = width, *item1Height = height*

item2Width = width, *item2Height = height*

The first object type is a transposed matrix. Therefore, `src1` stored in the memory has the following size

src1Width = item1Height = height,

src1Height = item1Width = width

and after the transposition, you will get the right first object size.

The second object type is a single matrix, therefore *src2Width* = *item2Width* = *width*,

src2Height = *item2Height* = *height*

Example 4:

```
IppStatus ippmMul_mm_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2,
   int matr1Width, int matr1Height, const Ipp32f* pSrc2, int src2Stride1,
   int src2Stride2, int matr2Width, int matr2Height, Ipp32f* pDst, int dstStride1,
   int dst2Stride2);
```

Object sizes follow each of the `src` matrices, therefore, these are the `src` sizes.

src1Width = *matr1Width*, *src1Height* = *matr1Height*

src2Width = *matr2Width*, *src2Height* = *matr2Height*

The function purpose is multiplication of matrices, therefore the width of the product is equal to the width of the second multiplier, and the height of the product is equal to the height of the first multiplier. The *dst* sizes must be

dstWidth = *matr2Width*, *dstHeight* = *matr1Height*

Example 5:

```
IppStatus ippmMul_tm_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2,
    int matr1Width, int matr1Height, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2,
    int matr2Width, int matr2Height, Ipp32f* pDst, int dstStride1, int dst2Stride2);
```

Object sizes follow each of the `src` matrices, therefore, these are the `src` sizes.

src1Width = *matr1Width*, *src1Height* = *matr1Height*

src2Width = *matr2Width*, *src2Height* = *matr2Height*

The function purpose is multiplication of matrices, therefore the width of the product is equal to the width of the second multiplier, and the height of the product is equal to the height of the first multiplier.

The first object type is a transposed matrix, while the second object type is an ordinary matrix. Therefore, the sizes of the multipliers are equal.

efficient1Width = *matr1Height*, *efficient1Height* = *matr1Width*

efficient2Width = *matr2Width*, *efficient2Height* = *matr2Height*

The *dst* sizes must be

dstWidth = *efficient2Width* = *matr2Width*,

dstHeight = *efficient1Height* = *matr1Width*

Error Reporting

The Intel IPP functions return the status of the performed operation to report errors and warnings to the calling program. Thus, it is up to the application to perform error-related actions and/or recover from the error. The last value of the error status is not stored, and the user is to decide whether to check it or not as the function returns. The status values are of IppStatus type and are global constant integers.

Below you can see a list of status codes and corresponding messages reported by the Intel IPP for small matrices.

`ippStsSizeMatchMatrixErr`	Unsuitable sizes of the source matrices.
`ippStsCountMatrixErr`	Count parameter is negative or equal to 0.
`ippStsRoiShiftMatrixErr`	RoiShift is negative or not divisible by the size of the data type.
`ippStsStrideMatrixErr`	Stride value is not positive or not divisible by the size of the data type.
`ippStsSingularErr`	Matrix is singular.
`ippStsNotPosDefErr`	Not positive-definite matrix.
`ippStsSizeErr`	Wrong value of the data size.
`ippStsNoErr`	No error, it's OK.
`ippStsDivByZeroErr`	An attempt to divide by zero.
`ippStsNullPtrErr`	Null pointer error.
`ippStsConvergeErr`	Indicates an error if the algorithm does not converge.
`ippStsSizeMatchMatrixErr`	Unsuitable sizes of the source matrices.

The status codes ending with `Err` (except for the `ippStsNoErr` status) indicate an error; the integer values of these codes are negative. When an error occurs, the function execution is interrupted.

For example, if the source matrix for `ippmLUDecomp` is singular, the function stops execution and returns with the error status `ippStsSingularErr`. If the input stride value is 0 or 3, the function stops execution and returns with the error status `ippStsStrideMatrixErr`.

Code Examples

The manual contains a number of code examples to demonstrate both some particular features of the small matrix functions and how these functions can be called.

Many of these code examples output result data together with status code and associated messages in case when error condition was met.

To keep the example code simpler, special definitions of print statements are used for better representation of results, as well as print status codes and messages.

The code definitions given below make it possible to build the examples contained in the manual by straightforward copying and pasting the example code fragments.

Example

Code Definitions

```
/*
// The functions providing simple output of the result
// for single precision and double precision real data.
// These functions are only for tight data:
//   Stride2 = sizeof(dataType)
//   Srtide1 = width*sizeof(dataType)
//   Stride0 = length*sizeof(dataType) - for vector array
//   Stride0 = width*height*sizeof(dataType) - for matrix array
*/

#define genPRINT_m(TYPE) \
void printf_m_Ipp##TYPE(const
char* msg, Ipp##TYPE* buf, \
                     int width, int height, IppStatus st ) \
{   int i, j; \
    if( st < ippStsNoErr ) { \
        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
    } else { \
        printf("%s \n", msg ); \
        for( j=0; j <  height; j++) { \
      for( i=0; i < width; i++) { \
             printf("%f  ", buf[j*width+i]); } \
      printf("\n"); } } \
}

#define genPRINT_ma(TYPE) \
void printf_ma_Ipp##TYPE(const char* msg, Ipp##TYPE## *buf, \
                     int width, int height, int count, IppStatus st ) \
{   int i, j, k; \
    if( st < ippStsNoErr ) { \
        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
    } else { \
        printf("%s \n", msg ); \
```

```
        for( j=0; j <  height; j++) { \
      for( k=0; k < count; k++) { \
                for( i=0; i < width; i++){ \
              printf("%f ", buf[j*width+i+k*width*height]); \
      } printf("    "); } printf("\n");}} \
}

#define genPRINT_m_L(TYPE) \
void printf_ma_Ipp##TYPE##_L(const
char* msg, Ipp##TYPE** buf, \
                     int width, int height, int count, IppStatus st )\
{   int i, j, k; \
    Ipp##TYPE## *dst;  \
    if( st < ippStsNoErr ) { \
        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
    } else { \
        printf("%s \n", msg ); \
        for( j=0; j <  height; j++) { \
      for( k=0; k < count; k++) { \
          dst = (Ipp##TYPE##*)buf[k]; \
                for( i=0; i < width; i++) { \
                 printf("%f ", dst[j*width+i]); } \
      printf("    "); } printf("\n"); } } \
}

#define genPRINT_m_P(TYPE) \
void printf_m_Ipp##TYPE##_P(const char* msg, Ipp##TYPE** buf, \
                       int width, int height, IppStatus st ) \
{   int i, j; \
    if( st < ippStsNoErr ) { \
        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
    } else { \
        printf("%s \n", msg ); \
        for( j=0; j <  height; j++) { \
            for( i=0; i < width; i++) { \
```

```
              printf("%f ", *buf[j*width+i]); } \
      printf("\n"); } } \
}

#define genPRINT_va(TYPE) \
void printf_va_Ipp##TYPE(const char* msg, Ipp##TYPE* buf, \
                     int length, int count, IppStatus st ) \
{   int i, j; \
    if( st < ippStsNoErr ) { \
        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
    } else { \
        printf("%s \n", msg ); \
        for( j=0; j <  count; j++) { \
      for( i=0; i < length; i++) { \
             printf("%f  ", buf[j*length+i]); } \
      printf("\n"); } } \
}

void printf_v_int(const char* msg, int* buf, int length) \
{   int i; \
    printf("%s \n", msg ); \
    for( i=0; i < length; i++) \
     printf("%d  ", buf[i]); \
    printf("\n"); \
}
genPRINT_va( 32f );
genPRINT_m( 32f );
genPRINT_ma( 32f );
genPRINT_m_P( 32f );
genPRINT_m_L( 32f );
genPRINT_va( 64f );
genPRINT_m( 64f );
genPRINT_ma( 64f );
genPRINT_m_P( 64f );
genPRINT_m_L( 64f );
```

Utility Functions

3

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) functions for small matrices that copy an object of any type to another object of any type, extract Regions of Interest (ROI), and initialize matrices.

Utility functions

Function Base Name	Operation
`Copy`	Performs copy operation.
`Extract`	Performs ROI extraction.
`LoadIdentity`	Initializes identity matrix.

Copy

Performs copy operation.

Syntax

Case 1: Vector array operation

```
IppStatus ippmCopy_va_32f_SS(const Ipp32f* pSrc, int srcStride0, int srcStride2, Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmCopy_va_64f_SS(const Ipp64f* pSrc, int srcStride0, int srcStride2, Ipp64f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmCopy_va_32f_SP(const Ipp32f* pSrc, int srcStride0, int srcStride2, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmCopy_va_64f_SP(const Ipp64f* pSrc, int srcStride0, int srcStride2, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmCopy_va_32f_SL(const Ipp32f* pSrc, int srcStride0, int srcStride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len, int count);

IppStatus ippmCopy_va_64f_SL(const Ipp64f* pSrc, int srcStride0, int srcStride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len, int count);

IppStatus ippmCopy_va_32f_LS(const Ipp32f** ppSrc, int srcRoiShift, int srcStride2, Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmCopy_va_64f_LS(const Ipp64f** ppSrc, int srcRoiShift, int srcStride2, Ipp64f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmCopy_va_32f_PS(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);
```

IppStatus ippmCopy_va_64f_PS(const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmCopy_va_32f_LP(const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmCopy_va_64f_LP(const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmCopy_va_32f_LL(const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmCopy_va_64f_LL(const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmCopy_va_32f_PP(const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmCopy_va_64f_PP(const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, Ipp64f** *pDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmCopy_va_32f_PL(const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmCopy_va_64f_PL(const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

Case 2: Matrix array operation

IppStatus ippmCopy_ma_32f_SS(const Ipp32f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmCopy_ma_64f_SS(const Ipp64f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmCopy_ma_32f_SP(const Ipp32f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmCopy_ma_64f_SP(const Ipp64f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmCopy_ma_32f_SL(const Ipp32f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmCopy_ma_64f_SL(const Ipp64f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmCopy_ma_32f_LS(const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride1*, int *srcStride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

```
IppStatus ippmCopy_ma_64f_LS(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1, int
srcStride2, Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int
height, int count);

IppStatus ippmCopy_ma_32f_PS(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, Ipp32f*
pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmCopy_ma_64f_PS(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0, Ipp64f*
pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmCopy_ma_32f_LP(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1, int
srcStride2, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int
count);

IppStatus ippmCopy_ma_64f_LP(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1, int
srcStride2, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int
count);

IppStatus ippmCopy_ma_32f_LL(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1, int
srcStride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width,
int height, int count);

IppStatus ippmCopy_ma_64f_LL(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1, int
srcStride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width,
int height, int count);

IppStatus ippmCopy_ma_32f_PP(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, Ipp32f**
ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmCopy_ma_64f_PP(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0, Ipp64f**
ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmCopy_ma_32f_PL(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, Ipp32f**
ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmCopy_ma_64f_PL(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0, Ipp64f**
ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source object or array of objects.
srcStride0	Stride between the objects in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source object.
srcRoiShift	ROI shift in the source object.
pDst, ppDst	Pointer to the destination object or array of objects.
dstStride0	Stride between the objects in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination object.
dstRoiShift	ROI shift in the destination object.
width	Matrix width.

height	Matrix height.
len	Vector length.
count	Number of objects in the array.

Description

The function `ippmCopy` is declared in the `ippm.h` header file. The function copies an object of any type to another object of any type and stores the result in the destination object.

If performed on matrices, all matrices involved in the operation must have the number of columns not less than *width* and the number of rows not less than *height*.

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the *roiShift* value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the *count* value is less or equal to zero.

Example

The code example below show how to use `ippmCopy_va_32f_PS` function.

```
IppStatus copy_va_32f_PS(void) {
    /* Source data:  */
    Ipp32f a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    /*
    // Pointer description for source data of interest a[0], a[6], a[7]:
    */
    Ipp32f* ppSrc[3] = { a, a+6, a+7 }; /* pointers array */
    int srcRoiShift = 0;

    int srcStride0  = sizeof(Ipp32f); /* formally must be initialized */
```

```
    /*
    // Standard description for destination vector
    */
    Ipp32f pDst[3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = sizeof(Ipp32f)*3;  /* formally must be initialized */

    int length = 3;
    int count  = 1;


    IppStatus status = ippmCopy_va_32f_PS((const Ipp32f**)ppSrc,

        srcRoiShift, srcStride0, pDst, dstStride0,

        dstStride2, length, count );


    /*
    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */
    if (status == ippStsNoErr) {
        printf_va_Ipp32f("Destination vector:", pDst, 3, 1, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }

    return status;
}
```

Output:

```
Destination vector:
0.000000  6.000000  7.000000
```

The code example below shows how to use ippmCopy_ma_32f_LS function.

```
IppStatus copy_ma_32f_LS(void) {
    /* Source data: 4 matrices with width=3 and height=2 */
    Ipp32f a[2*3] = { 10, 11, 12, 13, 14, 15 };
    Ipp32f b[2*3] = { 20, 21, 22, 23, 24, 25 };
```

```
    Ipp32f c[2*3] = { 30, 31, 32, 33, 34, 35 };
    Ipp32f d[2*3] = { 40, 41, 42, 43, 44, 45 };
    /*

    // Layout description for 4 source matrices:
    */
    Ipp32f* ppSrc[4] = { a, b, c, d }; /* matrix pointers array */
    int srcRoiShift = 0;
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = sizeof(Ipp32f)*3;
    /*
    // Standard description for 4 destination matrices
    */

    Ipp32f pDst[4*2*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = sizeof(Ipp32f)*3;
    int dstStride0 = sizeof(Ipp32f)*3*2;

    int width  = 3;
    int height = 2;
    int count  = 4;
    IppStatus status = ippmCopy_ma_32f_LS((const Ipp32f**)ppSrc,
        srcRoiShift, srcStride1, srcStride2, pDst, dstStride0,
        dstStride1, dstStride2, width, height, count );

    /*

    // It is recommended to check return status


    // to detect wrong input parameters, if any


    */

    if (status == ippStsNoErr) {

        printf_va_Ipp32f("4 destination matrices:", pDst, 2*3, 4, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }


  return status;
}
```

Output:

4 destination matrices:

```
10.000000  11.000000  12.000000  13.000000  14.000000  15.000000

20.000000  21.000000  22.000000  23.000000  24.000000  25.000000

30.000000  31.000000  32.000000  33.000000  34.000000  35.000000

40.000000  41.000000  42.000000  43.000000  44.000000  45.000000
```

Extract

Performs ROI extraction.

Syntax

Case 1: Vector operation

```
IppStatus ippmExtract_v_32f(const Ipp32f* pSrc, int srcStride2, Ipp32f* pDst, int len);

IppStatus ippmExtract_v_64f(const Ipp64f* pSrc, int srcStride2, Ipp64f* pDst, int len);

IppStatus ippmExtract_v_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f* pDst, int
len);

IppStatus ippmExtract_v_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f* pDst, int
len);
```

Case 2: Vector array operation

```
IppStatus, ippmExtract_va_32f(const Ipp32f* pSrc, int srcStride0, int srcStride2, Ipp32f*
pDst, int len, int count);

IppStatus ippmExtract_va_64f(const Ipp64f* pSrc, int srcStride0, int srcStride2, Ipp64f*
pDst, int len, int count);

IppStatus ippmExtract_va_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0,
Ipp32f* pDst, int len, int count);

IppStatus ippmExtract_va_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0,
Ipp64f* pDst, int len, int count);

IppStatus ippmExtract_va_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride2,
Ipp32f* pDst, int len, int count);

IppStatus ippmExtract_va_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride2,
Ipp64f* pDst, int len, int count);
```

Case 3: Matrix operation

```
IppStatus ippmExtract_m_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2, Ipp32f*
pDst, int width, int height);

IppStatus ippmExtract_m_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2, Ipp64f*
pDst, int width, int height);

IppStatus ippmExtract_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f* pDst, int
width, int height);

IppStatus ippmExtract_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f* pDst, int
width, int height);
```

Case 4: Transposed matrix operation

```
IppStatus ippmExtract_t_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2, Ipp32f*
pDst, int width, int height);

IppStatus ippmExtract_t_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2, Ipp64f*
pDst, int width, int height);
```

```
IppStatus ippmExtract_t_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f* pDst, int
width, int height);
```

```
IppStatus ippmExtract_t_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f* pDst, int
width, int height);
```

Case 5: Matrix array operation

```
IppStatus ippmExtract_ma_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1, int
srcStride2, Ipp32f* pDst, int width, int height, int count);
```

```
IppStatus ippmExtract_ma_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1, int
srcStride2, Ipp64f* pDst, int width, int height, int count);
```

```
IppStatus ippmExtract_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0,
Ipp32f* pDst, int width, int height, int count);
```

```
IppStatus ippmExtract_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0,
Ipp64f* pDst, int width, int height, int count);
```

```
IppStatus ippmExtract_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1, int
srcStride2, Ipp32f* pDst, int width, int height, int count);
```

```
IppStatus ippmExtract_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1, int
srcStride2, Ipp64f* pDst, int width, int height, int count);
```

Case 6: Transposed matrix array operation

```
IppStatus ippmExtract_ta_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1, int
srcStride2, Ipp32f* pDst, int width, int height, int count);
```

```
IppStatus ippmExtract_ta_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1, int
srcStride2, Ipp64f* pDst, int width, int height, int count);
```

```
IppStatus ippmExtract_ta_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0,
Ipp32f* pDst, int width, int height, int count);
```

```
IppStatus ippmExtract_ta_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0,
Ipp64f* pDst, int width, int height, int count);
```

```
IppStatus ippmExtract_ta_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1, int
srcStride2, Ipp32f* pDst, int width, int height, int count);
```

```
IppStatus ippmExtract_ta_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1, int
srcStride2, Ipp64f* pDst, int width, int height, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source object or array of objects.
srcStride0	Stride between the objects in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source object.
srcRoiShift	ROI shift in the source object.
pDst	Pointer to the specified destination object or array of objects.
len	Vector length.

width	Matrix width.
height	Matrix height.
count	Number of objects in the array.

Description

The function `ippmExtract` is declared in the `ippm.h` header file. The function extracts ROI from an object of any type to another object with specific properties.

When the operation is performed on vectors, the destination object is a dense vector or dense vector array.

When the operation is performed on matrices, the destination object is a dense matrix or a dense matrix array. The matrices involved in the operation must have the number of columns equal to *width* and the number of rows equal to *height*.

Note that if the operation is performed on a transposed matrix or an array of transposed matrices, the source matrices must have the number of columns equal to *height* and the number of rows equal to *width*.

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the *roiShift* value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the *count* value is less or equal to zero.

LoadIdentity

Initializes identity matrix.

Syntax

Case 1: Matrix array operation

```
IppStatus ippmLoadIdentity_ma_32f(const Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmLoadIdentity_ma_64f(const Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmLoadIdentity_ma_32f_P(const Ipp32f** ppDst, int dstRoiShift, int dstStride2, int width, int height, int count);

IppStatus ippmLoadIdentity_ma_64f_P(const Ipp64f** ppDst, int dstRoiShift, int dstStride2, int width, int height, int count);

IppStatus ippmLoadIdentity_ma_32f_L(Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmLoadIdentity_ma_64f_L(Ipp64f** ppDst, int dstRoiShift, int dstStride1, int
dstStride2, int width, int height, int count);
```

Parameters

pDst, *ppDst*	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix(ces).
dstStride2	Stride between the elements in the destination matrix(ces).
dstRoiShift	ROI shift in the destination matrix.
width	Matrix width.
height	Matrix height.
count	Number of objects in the array.

Description

The function `ippmLoadIdentity` is declared in the `ippm.h` header file. The function loads an identity matrix and stores the result in the destination object.

The destination matrix has the number of columns equal to *width* and the number of rows equal to *height*.

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is `NULL`.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the *roiShift* value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the *count* value is less or equal to zero.

Vector Algebra Functions

4

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) functions for small matrices that perform vector algebra operations.

Vector Algebra functions

Function Base Name	Operation
`Saxpy`	Performs the "saxpy" operation on vectors.
`Add`	Adds constant to vector or vector to another vector.
`Sub`	Subtracts constant from vector, vector from constant, or vector from another vector.
`Mul`	Multiplies vector by constant.
`CrossProduct`	Computes cross product of two 3D vectors.
`DotProduct`	Computes dot product of two vectors.
`L2Norm`	Computes vector's L2 norm.
`LComb`	Composes linear combination of two vectors.

Saxpy

Performs the "saxpy" operation on vectors.

Syntax

Case 1: Vector - vector operation

```
IppStatus ippmSaxpy_vv_32f(const Ipp32f* pSrc1, int src1Stride2, Ipp32f scale, const
Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst, int dstStride2, int len);
```

```
IppStatus ippmSaxpy_vv_64f(const Ipp64f* pSrc1, int src1Stride2, Ipp64f scale, const
Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst, int dstStride2, int len);
```

```
IppStatus ippmSaxpy_vv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, Ipp32f scale, const
Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int len);
```

```
IppStatus ippmSaxpy_vv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, Ipp64f scale, const
Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int len);
```

Case 2: Vector - vector array operation

```
IppStatus ippmSaxpy_vva_32f(const Ipp32f* pSrc1, int src1Stride2, Ipp32f scale, const
Ipp32f* pSrc2, int src2Stride0, int src2Stride2, Ipp32f* pDst, int dstStride0, int
dstStride2, int len, int count);
```

```
IppStatus ippmSaxpy_vva_64f(const Ipp64f* pSrc1, int src1Stride2, Ipp64f scale, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSaxpy_vva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, Ipp32f scale, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSaxpy_vva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, Ipp64f scale, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSaxpy_vva_32f_L(const Ipp32f* pSrc1, int src1Stride2, Ipp32f scale, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len, int count);

IppStatus ippmSaxpy_vva_64f_L(const Ipp64f* pSrc1, int src1Stride2, Ipp64f scale, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

Case 3: Vector array - vector operation

```
IppStatus ippmSaxpy_vav_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride2, Ipp32f scale, const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSaxpy_vav_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride2, Ipp64f scale, const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSaxpy_vav_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0, Ipp32f scale, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSaxpy_vav_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0, Ipp64f scale, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSaxpy_vav_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride2, Ipp32f scale, const Ipp32f* pSrc2, int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len, int count);

IppStatus ippmSaxpy_vav_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride2, Ipp64f scale, const Ipp64f* pSrc2, int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

Case 4: Vector array - vector array operation

```
IppStatus ippmSaxpy_vava_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride2, Ipp32f scale, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSaxpy_vava_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride2, Ipp64f scale, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride2, int len, int count);
```

```
IppStatus ippmSaxpy_vava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
Ipp32f scale, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst,
int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSaxpy_vava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
Ipp64f scale, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst,
int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSaxpy_vava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride2,
Ipp32f scale, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp32f** ppDst,
int dstRoiShift, int dstStride2, int len, int count);

IppStatus ippmSaxpy_vava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride2,
Ipp64f scale, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp64f** ppDst,
int dstRoiShift, int dstStride2, int len, int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the first source vector or array.
src1Stride0	Stride between the vectors in the first source vector array.
src1Stride2	Stride between the elements in the first source vector(s).
src1RoiShift	ROI shift in the first source vector(s).
pSrc2, ppSrc2	Pointer to the second source vector or vector array.
src2Stride0	Stride between the vectors in the second source vector array.
src2Stride2	Stride between the elements in the second source vector(s).
src2RoiShift	ROI shift in the second source vector(s).
pDst, ppDst	Pointer to the destination vector or vector array.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector(s).
dstRoiShift	ROI shift in the destination vector(s).
scale	Multiplier.
len	Vector length.
count	Number of vectors in the array.

Description

The function `ippmSaxpy` is declared in the `ippm.h` header file. The function composes linear combination of two vectors by multiplying the first source vector by a constant, adding it to the second vector, and storing the result in the destination vector:

$$dst[i] = scale \times src1[i] + src2[i],\ 0 \le i < len.$$

Return Values

`ippStsOk`	Returns no error.

`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

Example

The code example below shows how to use `ippmSaxpy_vav_32f` function.

```
IppStatus saxpy_vav_32f(void) {
    /* Src1 is 2 vectors with length=4 */
    Ipp32f pSrc1[2*4] = { 1, 2, 4, 8,
                          3, 5, 7, 9};
    /* Src2 is vector with length=4 */
    Ipp32f pSrc2[4] = { -1, -5, -2 , -3 };

    Ipp32f scale = 2.0;
    /* Standard description for source vectors */
    int src1Stride2 = sizeof(Ipp32f);
    int src1Stride0 = 4*sizeof(Ipp32f);
    int src2Stride2 = sizeof(Ipp32f);

    /* Standard description for destination vectors */
    Ipp32f pDst[2*4];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = 4*sizeof(Ipp32f);
    int length = 4;
    int count  = 2;
    IppStatus status = ippmSaxpy_vav_32f((const Ipp32f*)pSrc1,
        src1Stride0, src1Stride2, scale,(const Ipp32f*)pSrc2,
         src2Stride2, pDst, dstStride0, dstStride2, length, count);
    /*
    // It is recommended to check return status

    // to detect wrong input parameters, if any
```

```
    */
    if (status == ippStsNoErr) {
        printf_va_Ipp32f("Dst is 2 vectors:", pDst, 4, 2, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }

    return status;
}
```

Output:

```
Dst is 2 vectors:
1.000000  -1.000000  6.000000  13.000000
5.000000  5.000000  12.000000  15.000000
```

Add

Adds constant to vector or vector to another vector.

Syntax

Case 1: Vector - constant operation

IppStatus ippmAdd_vc_32f(const Ipp32f* *pSrc*, int *srcStride2*, Ipp32f *val*, Ipp32f* *pDst*, int *dstStride2*, int *len*);

IppStatus ippmAdd_vc_64f(const Ipp64f* *pSrc*, int *srcStride2*, Ipp64f *val*, Ipp64f* *pDst*, int *dstStride2*, int *len*);

IppStatus ippmAdd_vc_32f_P(const Ipp32f** *ppSrc*, int *srcRoiShift*, Ipp32f *val*, Ipp32f** *ppDst*, int *dstRoiShift*, int *len*);

IppStatus ippmAdd_vc_64f_P(const Ipp64f** *ppSrc*, int *srcRoiShift*, Ipp64f *val*, Ipp64f** *ppDst*, int *dstRoiShift*, int *len*);

Case 2: Vector array - constant operation

IppStatus ippmAdd_vac_32f(const Ipp32f* *pSrc*, int *srcStride0*, int *srcStride2*, Ipp32f *val*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vac_64f(const Ipp64f* *pSrc*, int *srcStride0*, int *srcStride2*, Ipp64f *val*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vac_32f_P(const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, Ipp32f *val*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmAdd_vac_64f_P(const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, Ipp64f *val*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmAdd_vac_32f_L(const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride2*, Ipp32f *val*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vac_64f_L(const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride2*, Ipp64f *val*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

Case 3: Vector array - constant array operation

IppStatus ippmAdd_vaca_32f (const Ipp32f* *pSrc*, int *srcStride0*, int *srcStride2*, const Ipp32f* *pVal*, int *valStride0*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vaca_64f (const Ipp64f* *pSrc*, int *srcStride0*, int *srcStride2*, const Ipp64f* *pVal*, int *valStride0*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vaca_32f_P (const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, const Ipp32f* *pVal*, int *valStride0*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmAdd_vaca_64f_P (const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, const Ipp64f* *pVal*, int *valStride0*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmAdd_vaca_32f_L (const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride2*, const Ipp32f** *ppVal*, int *valRoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vaca_64f_L (const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride2*, const Ipp64f** *ppVal*, int *valRoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

Case 4: Vector - constant array operation

IppStatus ippmAdd_vca_32f (const Ipp32f* *pSrc*, int *srcStride2*, const Ipp32f* *pVal*, int *valStride0*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vca_64f (const Ipp64f* *pSrc*, int *srcStride2*, const Ipp64f* *pVal*, int *valStride0*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vca_32f_P (const Ipp32f** *ppSrc*, int *srcRoiShift*, const Ipp32f* *pVal*, int *valStride0*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmAdd_vca_64f_P (const Ipp64f** *ppSrc*, int *srcRoiShift*, const Ipp64f* *pVal*, int *valStride0*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmAdd_vca_32f_L (const Ipp32f* *pSrc*, int *srcStride2*, const Ipp32f** *ppVal*, int *valRoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vca_64f_L (const Ipp64f* *pSrc*, int *srcStride2*, const Ipp64f** *ppVal*, int *valRoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

Case 5: Vector array - vector operation

IppStatus ippmAdd_vv_32f(const Ipp32f* *pSrc1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride2*, int *len*);

IppStatus ippmAdd_vv_64f(const Ipp64f* *pSrc1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride2*, int *len*);

IppStatus ippmAdd_vv_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *len*);

IppStatus ippmAdd_vv_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *len*);

Case 6: Vector array - vector operation

IppStatus ippmAdd_vav_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vav_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vav_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmAdd_vav_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmAdd_vav_32f_L(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vav_64f_L(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

Case 7: Vector array - vector array operation

IppStatus ippmAdd_vava_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride0*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vava_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride0*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmAdd_vava_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmAdd_vava_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmAdd_vava_32f_L(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride2*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

```
IppStatus ippmAdd_vava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride2,
const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
int dstStride2, int len, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source vector or vector array.
srcStride0	Stride between the vectors in the source array.
srcStride2	Stride between the elements in the source vector(s).
srcRoiShift	ROI shift in the first source vector.
pSrc1, ppSrc1	Pointer to the first source vector or vector array.
src1Stride0	Stride between the vectors in the first source vector array.
src1Stride2	Stride between the elements in the first source vector(s).
src1RoiShift	ROI shift in the first source vector.
pSrc2, ppSrc2	Pointer to the second source vector or vector array.
src2Stride0	Stride between the vectors in the second source vector array.
src2Stride2	Stride between the elements in the second source vector(s).
src2RoiShift	ROI shift in the second source vector(s).
val	The constant.
pVal, ppVal	Pointer to the source array of constants.
valStride0	Stride between the constants in the source array of constants.
valRoiShift	ROI shift in the source array of constants.
pDst, ppDst	Pointer to the destination vector or vector array.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector.
dstRoiShift	ROI shift in the destination vector.
len	Vector length.
count	The number of vectors (constants) in the array.

Description

The function ippmAdd is declared in the ippm.h header file. Like all other functions in Intel IPP for small matrices, this function is parameter sensitive. All input parameters that follow the function name immediately after the underscore determine the way in which the function performs and the arguments it takes, whether it is a constant or another vector. This implies that with every complete function name, only some of the listed arguments appear in the input list, while others are omitted.

When performed on a constant together with a vector, the function adds *val* to each element of the source vector and stores the result into destination vector:

$$dst[i] = val + src[i],\ 0 \le i < len.$$

To clarify how the function operates on arrays of vectors and constants, see the "Operations with arrays of objects" section in Chapter 2.

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

Example

The code example below demonstrates how to use the function ippmAdd_vc_32f_P.

```
IppStatus add_vc_32f_P(void) {
    /* Source data:  */
    Ipp32f a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    Ipp32f val = 10.0;
    /*
    // Pointer description for source data of interest a[0], a[6], a[7]:
    */
    Ipp32f* ppSrc[3] = { a, a+6, a+7 }; /* pointers array */
    int srcRoiShift = 0;


    /*
    // Pointer description for destination data of interest a[0], a[6], a[7]:
    */
    Ipp32f* ppDst[3] = { a, a+6, a+7 }; /* pointers array */
    int dstRoiShift  = 0;

    int length = 3;

    IppStatus status = ippmAdd_vc_32f_P((const Ipp32f**)ppSrc,
        srcRoiShift, val, ppDst, dstRoiShift, length);

    /*
    // It is recommended to check return status
    // to detect wrong input parameters, if any
    */
    if (status == ippStsNoErr){
        printf_va_Ipp32f("Destination vector:", a, 10, 1, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
    return status;
}
```

Output:

```
Destination vector:

10.000000  1.000000  2.000000  3.000000  4.000000  5.000000  16.000000  17.000000  8.000000  9.000000
```

The code examples below illustrate the use of the ippmAdd function operating on arrays of constants.

The code example below demonstrates how to use function ippmAdd_vaca_32f when all elements in the data vectors, all vectors and constants in the arrays are regularly spaced in memory.

```
IppStatus add_vaca_32f(void)  {

    /* Src data: 4 vectors on length=3, vector elements
    are spaced with step */
    Ipp32f pSrc[4*6] = { 1, 0, 2, 0, 3, 0,
                         4, 0, 5, 0, 6, 0,
                         7, 0, 8, 0, 9, 0,
                        10, 0,11, 0,12, 0 };
    /* Standart description for Src vector array */
    int srcStride2 = 2*sizeof(Ipp32f);
    int srcStride0 = 3*2*sizeof(Ipp32f);
    /* Standart description for Val constant array */
    Ipp32f pVal[9] = { 10, 0, 0, 9, 0, 0, 8, 0, 0};
    int valStride0 = 3*sizeof(Ipp32f);

    /* Standard description for Dst vector array*/
    Ipp32f pDst[4*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = 3*sizeof(Ipp32f);

    int length = 3;
    int count  = 4;

    IppStatus status = ippmAdd_vaca_32f ((const Ipp32f*)pSrc,
        srcStride0, srcStride2, (const Ipp32f*)pVal, valStride0,
        pDst, dstStride0, dstStride2, length, count);

    /*
    // It is recommended to check return status
    // to detect wrong input parameters, if any
    */
    if(status == ippStsOk){
        printf_va_Ipp32f("4 destination vectors:", pDst, 3, 4, status);

    } else {
       printf("Function returns status: %s \n", ippGetStatusString(status));

    }

    return status;
}
```

Output:

```
4 destination vectors:

11.000000  12.000000  13.000000

13.000000  14.000000  15.000000

15.000000  16.000000  17.000000

10.000000  11.000000  12.000000
```

The code example below demonstrates how to use the function `ippmAdd_vaca_32f_L` when vector elements are regularly spaced but vectors and constants in the arrays are irregularly spaced in memory.

```
IppStatus add_vaca_32f_L (void) {
    /* Src data: 4 vectors with length=3, Stride2=2*sizeof(Ipp32f) */
    Ipp32f src_a[2*3] = { 1, 0, 2, 0, 3, 0 };
    Ipp32f src_b[2*3] = { 4, 0, 5, 0, 6, 0 };
    Ipp32f src_c[2*3] = { 7, 0, 8, 0, 9, 0 };
    Ipp32f src_d[2*3] = {10, 0,11, 0,12, 0 };


    /* Layout description for Src */
    Ipp32f* ppSrc[4] = { src_a, src_b, src_c, src_d };
    int srcRoiShift = 0;
    int srcStride2 = 2*sizeof(Ipp32f);

    /* Val data array */
    Ipp32f pVal[10] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

    /*
    // Layout description for Val
    // 4 constants with irregular layout
    */
    Ipp32f* ppVal[4] = { pVal, pVal+1, pVal+3, pVal+9 };
    int valRoiShift = 0;

    /* Destination memory location */
    Ipp32f dst[4*3];

    /* Layout description for Dst */
    Ipp32f* ppDst[4] = { dst, dst+3, dst+6, dst+9 };
    int dstRoiShift = 0;
    int dstStride2 = sizeof(Ipp32f);

    int length = 3;
    int count  = 4;


    IppStatus status = ippmAdd_vaca_32f_L ((const Ipp32f**)ppSrc,
        srcRoiShift, srcStride2, (const Ipp32f**)ppVal, valRoiShift,
        ppDst, dstRoiShift, dstStride2, length, count);

    /*
    // It is recommended to check return status
    // to detect wrong input parameters, if any
    */
    if(status == ippStsOk){
        printf_va_Ipp32f("4 destination vectors:", dst, 3, 4, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
    return status;
}
```

Output:

```
4 destination vectors:

11.000000  12.000000  13.000000

13.000000  14.000000  15.000000
```

```
14.000000  15.000000  16.000000

11.000000  12.000000  13.000000
```

The code example below demonstrates how to use the function `ippmAdd_vca_32f_P` when vector elements are irregularly spaced but constants in the arrays are regularly spaced in memory.

```
IppStatus add_vca_32f_P (void) {

    /* Source data location */
    Ipp32f src[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    Ipp32f pVal[4] = {2.0, 5.0, 8.0, 10.0};

    /*
    // Pointer description for source data of interest a[0], a[6], a[7]:
    // Src is a vector with irregular layout, length=3
    */
    Ipp32f* ppSrc[3] = { src, src+6, src+7 };
    int srcRoiShift = 0;

    /*
    // Standard description for array of constants
    */
    int valStride0 = sizeof(Ipp32f);

    /* Destination memory location */
    Ipp32f dst[3*4];


    /* Pointer description for the destination vector */
    Ipp32f* ppDst[3] = { dst, dst+4, dst+8 };
    int dstRoiShift  = 0;
    int dstStride0   = sizeof(Ipp32f);

    int length = 3;
    int count = 4;

    IppStatus status = ippmAdd_vca_32f_P ((const Ipp32f**)ppSrc,
        srcRoiShift, (const Ipp32f*)pVal, valStride0, ppDst,
        dstRoiShift, dstStride0, length, count);

    /*
    // It is recommended to check return status
    // to detect wrong input parameters, if any
    */
    if(status == ippStsNoErr){
       printf_va_Ipp32f("4 destination vectors:", dst, 3, 4, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
    return status;
}
```

Output:

```
4 destination vectors:

 2.000000  5.000000  8.000000

10.000000  8.000000 11.000000
```

```
14.000000 16.000000  9.000000

12.000000 15.000000 17.000000
```

When performed on two vectors, the function adds together the respective elements of the first and the second source vectors and stores the result in the destination vector:

$$dst[i] = src1[i] + src2[i],\ 0 \le i < len.$$

The code example below demonstrates how to use the function `ippmAdd_vava_32f_L`.

```
IppStatus add_vava_32f_L(void) {
    /* Src1 data: 4 vectors with length=3, Stride2=2*sizeof(Ipp32f) */
    Ipp32f src1_a[2*3] = { 1, 0, 2, 0, 3, 0 };
    Ipp32f src1_b[2*3] = { 4, 0, 5, 0, 6, 0 };
    Ipp32f src1_c[2*3] = { 7, 0, 8, 0, 9, 0 };
    Ipp32f src1_d[2*3] = { 10, 0, 11, 0, 12, 0 };


    /* Src2 data: 4 vectors with length=3, Stride2=sizeof(Ipp32f) */
    Ipp32f src2_a[3] = { 10, 11, 12 };
    Ipp32f src2_b[3] = { 7, 8, 9 };
    Ipp32f src2_c[3] = { 4, 5, 6 };
    Ipp32f src2_d[3] = { 1, 2, 3 };

    /* Layout description for Src1, Src2 and Dst */
    Ipp32f* ppSrc1[4] = { src1_a, src1_b, src1_c, src1_d };
    Ipp32f* ppSrc2[4] = { src2_d, src2_c, src2_b, src2_a };
    int src1RoiShift = 0;
    int src2RoiShift = 0;

    Ipp32f dst[4*3]; /* destination memory location */
    Ipp32f* ppDst[4] = { dst, dst+3, dst+6, dst+9 };
    int dstRoiShift  = 0;

    int src1Stride2 = 2*sizeof(Ipp32f);
    int src2Stride2 = sizeof(Ipp32f);
    int dstStride2  = sizeof(Ipp32f);

    int length = 3;
    int count  = 4;

    IppStatus status = ippmAdd_vava_32f_L((const Ipp32f**)ppSrc1,
        src1RoiShift, src1Stride2, (const Ipp32f**)ppSrc2, src2RoiShift,
        src2Stride2, ppDst, dstRoiShift, dstStride2, length, count);

    /*
    // It is recommended to check return status
    // to detect wrong input parameters, if any
    */
    if (status == ippStsNoErr){
        printf_va_Ipp32f("4 destination vectors:", dst, 3, 4, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
    return status;
}
```

Output:

```
4 destination vectors:
 2.000000  4.000000  6.000000
 8.000000 10.000000 12.000000
14.000000 16.000000 18.000000
20.000000 22.000000 24.000000
```

Sub

Subtracts constant from vector, vector from constant, or vector from another vector.

Syntax

Case 1: Vector - constant operation

```
IppStatus ippmSub_vc_32f(const Ipp32f* pSrc, int srcStride2, Ipp32f val, Ipp32f* pDst, int dstStride2, int len);

IppStatus ippmSub_vc_64f(const Ipp64f* pSrc, int srcStride2, Ipp64f val, Ipp64f* pDst, int dstStride2, int len);

IppStatus ippmSub_vc_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int len);

IppStatus ippmSub_vc_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int len);
```

Case 2: Vector array - constant operation

```
IppStatus ippmSub_vac_32f(const Ipp32f* pSrc, int srcStride0, int srcStride2, Ipp32f val, Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSub_vac_64f(const Ipp64f* pSrc, int srcStride0, int srcStride2, Ipp64f val, Ipp64f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSub_vac_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSub_vac_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSub_vac_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride2, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len, int count);

IppStatus ippmSub_vac_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride2, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

Case 3: Vector array - constant array operation

```
IppStatus ippmSub_vaca_32f (const Ipp32f* pSrc, int srcStride0, int srcStride2, const Ipp32f* pVal, int valStride0, Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);
```

IppStatus ippmSub_vaca_64f (const Ipp64f* *pSrc*, int *srcStride0*, int *srcStride2*, const Ipp64f* *pVal*, int *valStride0*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmSub_vaca_32f_P (const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, const Ipp32f* *pVal*, int *valStride0*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmSub_vaca_64f_P (const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, const Ipp64f* *pVal*, *valStride0*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmSub_vaca_32f_L (const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride2*, const Ipp32f** *ppVal*, int *valRoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmSub_vaca_64f_L (const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride2*, const Ipp64f** *ppVal*, int *valRoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

Case 4: Vector - constant array operation

IppStatus ippmSub_vca_32f (const Ipp32f* *pSrc*, int *srcStride2*, const Ipp32f* *pVal*, int *valStride0*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmSub_vca_64f (const Ipp64f* *pSrc*, int *srcStride2*, const Ipp64f* *pVal*, int *valStride0*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmSub_vca_32f_P (const Ipp32f** *ppSrc*, int *srcRoiShift*, const Ipp32f* *pVal*, int *valStride0*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmSub_vca_64f_P (const Ipp64f** *ppSrc*, int *srcRoiShift*, const Ipp64f* *pVal*, int *valStride0*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmSub_vca_32f_L (const Ipp32f* *pSrc*, int *srcStride2*, const Ipp32f** *ppVal*, int *valRoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmSub_vca_64f_L (const Ipp64f* *pSrc*, int *srcStride2*, const Ipp64f** *ppVal*, int *valRoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

Case 5: Constant - vector operation

IppStatus ippmSub_cv_32f(const Ipp32f* *pSrc*, int *srcStride2*, Ipp32f *val*, Ipp32f* *pDst*, int *dstStride2*, int *len*);

IppStatus ippmSub_cv_64f(const Ipp64f* *pSrc*, int *srcStride2*, Ipp64f *val*, Ipp64f* *pDst*, int *dstStride2*, int *len*);

IppStatus ippmSub_cv_32f_P(const Ipp32f** *ppSrc*, int *srcRoiShift*, Ipp32f *val*, Ipp32f** *ppDst*, int *dstRoiShift*, int *len*);

IppStatus ippmSub_cv_64f_P(const Ipp64f** *ppSrc*, int *srcRoiShift*, Ipp64f *val*, Ipp64f** *ppDst*, int *dstRoiShift*, int *len*);

Case 6: Constant - vector array operation

IppStatus ippmSub_cva_32f(const Ipp32f* *pSrc*, int *srcStride0*, int *srcStride2*, Ipp32f *val*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

```
IppStatus ippmSub_cva_64f(const Ipp64f* pSrc, int srcStride0, int srcStride2, Ipp64f val,
Ipp64f* pDst, int dstStride0, int dstStride2, int len, int count);
```

```
IppStatus ippmSub_cva_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, Ipp32f
val, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len, int count);
```

```
IppStatus ippmSub_cva_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0, Ipp64f
val, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int count);
```

```
IppStatus ippmSub_cva_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride2, Ipp32f
val, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

```
IppStatus ippmSub_cva_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride2, Ipp64f
val, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

Case 7: Constant array - vector array operation

```
IppStatus ippmSub_cava_32f (const Ipp32f* pSrc, int srcStride0, int srcStride2, const
Ipp32f* pVal, int valStride0, Ipp32f* pDst, int dstStride0, int dstStride2, int len, int
count);
```

```
IppStatus ippmSub_cava_64f (const Ipp64f* pSrc, int srcStride0, int srcStride2, const
Ipp64f* pVal, int valStride0, Ipp64f* pDst, int dstStride0, int dstStride2, int len, int
count);
```

```
IppStatus ippmSub_cava_32f_P (const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, const
Ipp32f* pVal, int valStride0, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len,
int count);
```

```
IppStatus ippmSub_cava_64f_P (const Ipp64f** ppSrc, int srcRoiShift, int srcStride0, const
Ipp64f* pVal, int valStride0, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len,
int count);
```

```
IppStatus ippmSub_cava_32f_L (const Ipp32f** ppSrc, int srcRoiShift, int srcStride2, const
Ipp32f** ppVal, int valRoiShift, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len,
int count);
```

```
IppStatus ippmSub_cava_64f_L (const Ipp64f** ppSrc, int srcRoiShift, int srcStride2, const
Ipp64f** ppVal, int valRoiShift, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len,
int count);
```

Case 8: Constant array - vector operation

```
IppStatus ippmSub_cav_32f (const Ipp32f* pSrc, int srcStride2, const Ipp32f* pVal, int
valStride0, Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);
```

```
IppStatus ippmSub_cav_64f (const Ipp64f* pSrc, int srcStride2, const Ipp64f* pVal, int
valStride0, Ipp64f* pDst, int dstStride0, int dstStride2, int len, int count);
```

```
IppStatus ippmSub_cav_32f_P (const Ipp32f** ppSrc, int srcRoiShift, const Ipp32f* pVal,
int valStride0, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len, int count);
```

```
IppStatus ippmSub_cav_64f_P (const Ipp64f** ppSrc, int srcRoiShift, const Ipp64f* pVal,
int valStride0, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int count);
```

```
IppStatus ippmSub_cav_32f_L (const Ipp32f* pSrc, int srcStride2, const Ipp32f** ppVal,
int valRoiShift, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

IppStatus ippmSub_cav_64f_L (const Ipp64f* *pSrc*, int *srcStride2*, const Ipp64f** *ppVal*, int *valRoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

Case 9: Vector - vector operation

IppStatus ippmSub_vv_32f(const Ipp32f* *pSrc1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride2*, int *len*);

IppStatus ippmSub_vv_64f(const Ipp64f* *pSrc1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride2*, int *len*);

IppStatus ippmSub_vv_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *len*);

IppStatus ippmSub_vv_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *len*);

Case 10: Vector - vector array operation

IppStatus ippmSub_vva_32f(const Ipp32f* *pSrc1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride0*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmSub_vva_64f(const Ipp64f* *pSrc1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride0*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmSub_vva_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, const Ipp32f** *ppSrc2*, int *src2RoiShift* , int *src2Stride0*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmSub_vva_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

IppStatus ippmSub_vva_32f_L(const Ipp32f* *pSrc1*, int *src1Stride2*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmSub_vva_64f_L(const Ipp64f* *pSrc1*, int *src1Stride2*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *len*, int *count*);

Case 11: Vector array - vector operation

IppStatus ippmSub_vav_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmSub_vav_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmSub_vav_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

```
IppStatus ippmSub_vav_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int dstStride0,
int len, int count);
```

```
IppStatus ippmSub_vav_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride2,
const Ipp32f* pSrc2, int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2,
int len, int count);
```

```
IppStatus ippmSub_vav_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride2,
const Ipp64f* pSrc2, int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2,
int len, int count);
```

Case 12: Vector array - vector array operation

```
IppStatus ippmSub_vava_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride2, const
Ipp32f* pSrc2, int src2Stride0, int src2Stride2, Ipp32f* pDst, int dstStride0, int
dstStride2, int len, int count);
```

```
IppStatus ippmSub_vava_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride2, const
Ipp64f* pSrc2, int src2Stride0, int src2Stride2, Ipp64f* pDst, int dstStride0, int
dstStride2, int len, int count);
```

```
IppStatus ippmSub_vava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst, int dstRoiShift,
int dstStride0, int len, int count);
```

```
IppStatus ippmSub_vava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst, int dstRoiShift,
int dstStride0, int len, int count);
```

```
IppStatus ippmSub_vava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride2,
const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp32f** ppDst, int dstRoiShift,
int dstStride2, int len, int count);
```

```
IppStatus ippmSub_vava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride2,
const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
int dstStride2, int len, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source vector or vector array.
srcStride0	Stride between the vectors in the source array.
srcStride2	Stride between the elements in the source vector(s).
srcRoiShift	ROI shift in the source vector.
pSrc1, ppSrc1	Pointer to the first source vector or vector array.
src1Stride0	Stride between the vectors in the first source vector array.
src1Stride2	Stride between the elements in the first source vector(s).
src1RoiShift	ROI shift in the first source vector.
pSrc2, ppSrc2	Pointer to the second source vector or vector array.
src2Stride0	Stride between the vectors in the second source vector array.
src2Stride2	Stride between the elements in the second source vector(s).

src2RoiShift	ROI shift in the second source vector(s).
val	The constant.
pVal, ppVal	Pointer to the source array of constants.
valStride0	Stride between the constants in the source array of constants.
valRoiShift	ROI shift for the source array of constants.
pDst, ppDst	Pointer to the destination vector or vector array.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector.
dstRoiShift	ROI shift in the destination vector.
len	Vector length.
count	The number of vectors (constants) in the array.

Description

The function ippmSub is declared in the ippm.h header file. Like all other Intel IPP matrix operating functions, this function is parameter sensitive. All input parameters that follow the function name immediately after the underscore determine the way in which the function performs and the arguments it takes, whether it is a constant or another vector. This implies that with every complete function name only some of the listed arguments appear in the input list, while others are omitted.

When the function is performed on a vector and a constant, it subtracts *val* from each element of the source vector and stores the result in the destination vector:

$$dst[i] = src[i] - val,\ 0 \le i < len.$$

When the function is performed on a constant and a vector, it subtracts each element of the source vector from *val* and stores the result in the destination vector:

$$dst[i] = val - src[i],\ 0 \le i < len.$$

To clarify how the function operates on arrays of vectors and constants, see the "Operations with arrays of objects" section in Chapter 2.

When the function is performed on two vectors, it subtracts the elements of the second source vector from the respective elements of the first source vector and stores the result in the destination vector:

$$dst[i] = src1[i] - src2[i],\ 0 \le i < len.$$

To clarify how the function operates on two arrays of vectors, see the "Operations with arrays of objects" section in Chapter 2.

Examples of calling the function ippmSub operating with arrays of constants are similar to those of the function ippmAdd (see Example “ippmAdd_vaca_32f”, Example “ippmAdd_vaca_32f_L”, and Example “ippmAdd_vca_32f_P”).

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

Example

The code example below demonstrates how to use the function `ippmSub_vav_32f_P`.

```
IppStatus sub_vav_32f_P(void) {
    /* Source data */
    Ipp32f src1[3*4] = { 11, 21, 31, 41,
                         12, 22, 32, 42,
                         13, 23, 33, 43 };

    Ipp32f src2[3] = { 1, 2, 3 };
    /*
    // The first operand is 4 vector-columns with length=3
    // Pointer description:
    // ppSrc1[0] pointer to the first element of the first column
    // ppSrc1[1] pointer to the second element of the first column
    // ppSrc1[2] pointer to the third element  of the first column
    */
    Ipp32f* ppSrc1[3] = { src1, src1+4, src1+8 };
    int src1RoiShift = 0;
    int src1Stride0 = sizeof(Ipp32f); /* Stride between columns */

    /*
    // The second operand is vector with length=3
    // Pointer description:
    // ppSrc2[0] pointer to the first element
    // ppSrc2[1] pointer to the second element
    // ppSrc2[2] pointer to the third element
    */
    Ipp32f* ppSrc2[3] = { src2, src2+1, src2+2 };
    int src2RoiShift = 0;
    /*
    // Destination is 4 vectors with length=3
    // Pointer description for destination vectors
    */
    Ipp32f dst[12];
    int length = 3;
```

```
    int count  = 4;



    Ipp32f* ppDst[3] = { dst, dst+1, dst+2 };
    int dstRoiShift = 0;
    int dstStride0 = sizeof(Ipp32f)*3; /* Stride between vectors */

    IppStatus status = ippmSub_vav_32f_P((const Ipp32f**)ppSrc1,
        src1RoiShift, src1Stride0, (const Ipp32f**)ppSrc2, src2RoiShift,
        ppDst, dstRoiShift, dstStride0, length, count);
    /*
    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */
    if (status == ippStsNoErr) {
        printf_va_Ipp32f("Dst is 4 vectors:", dst, 3, 4, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }

    return status;
}
```

Output:

```
Dst is 4 vectors:
10.000000  10.000000  10.000000
20.000000  20.000000  20.000000
30.000000  30.000000  30.000000
40.000000  40.000000  40.000000
```

Mul

Multiplies vector by constant.

Syntax

Case 1: Vector - constant operation

```
IppStatus ippmMul_vc_32f(const Ipp32f* pSrc, int srcStride2, Ipp32f val, Ipp32f* pDst,
int dstStride2, int len);

IppStatus ippmMul_vc_64f(const Ipp64f* pSrc, int srcStride2, Ipp64f val, Ipp64f* pDst,
int dstStride2, int len);

IppStatus ippmMul_vc_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f val, Ipp32f**
ppDst, int dstRoiShift, int len);

IppStatus ippmMul_vc_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f val, Ipp64f**
ppDst, int dstRoiShift, int len);
```

Case 2: Vector array - constant operation

```
IppStatus ippmMul_vac_32f(const Ipp32f* pSrc, int srcStride0, int srcStride2, Ipp32f val,
Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmMul_vac_64f(const Ipp64f* pSrc, int srcStride0, int srcStride2, Ipp64f val,
Ipp64f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmMul_vac_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, Ipp32f
val, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmMul_vac_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0, Ipp64f
val, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmMul_vac_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride2, Ipp32f
val, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len, int count);

IppStatus ippmMul_vac_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride2, Ipp64f
val, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

Case 3: Vector array - constant array operation

```
IppStatus ippmMul_vaca_32f (const Ipp32f* pSrc, int srcStride0, int srcStride2, const
Ipp32f* pVal, int valStride0, Ipp32f* pDst, int dstStride0, int dstStride2, int len, int
count);

IppStatus ippmMul_vaca_64f (const Ipp64f* pSrc, int srcStride0, int srcStride2, const
Ipp64f* pVal, int valStride0, Ipp64f* pDst, int dstStride0, int dstStride2, int len, int
count);

IppStatus ippmMul_vaca_32f_P (const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, const
Ipp32f* pVal, int valStride0, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len,
int count);

IppStatus ippmMul_vaca_64f_P (const Ipp64f** ppSrc, int srcRoiShift, int srcStride0, const
Ipp64f* pVal, valStride0, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int
count);
```

```
IppStatus ippmMul_vaca_32f_L (const Ipp32f** ppSrc, int srcRoiShift, int srcStride2, const
Ipp32f** ppVal, int valRoiShift, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len,
int count);

IppStatus ippmMul_vaca_64f_L (const Ipp64f** ppSrc, int srcRoiShift, int srcStride2, const
Ipp64f** ppVal, int valRoiShift, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len,
int count);
```

Case 4: Vector - constant array operation

```
IppStatus ippmMul_vca_32f (const Ipp32f* pSrc, int srcStride2, const Ipp32f* pVal, int
valStride0, Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmMul_vca_64f (const Ipp64f* pSrc, int srcStride2, const Ipp64f* pVal, int
valStride0, Ipp64f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmMul_vca_32f_P (const Ipp32f** ppSrc, int srcRoiShift, const Ipp32f* pVal,
int valStride0, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmMul_vca_64f_P (const Ipp64f** ppSrc, int srcRoiShift, const Ipp64f* pVal,
int valStride0, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmMul_vca_32f_L (const Ipp32f* pSrc, int srcStride2, const Ipp32f** ppVal,
int valRoiShift, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len, int count);

IppStatus ippmMul_vca_64f_L (const Ipp64f* pSrc, int srcStride2, const Ipp64f** ppVal,
int valRoiShift, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

Parameters

pSrc, *ppSrc*	Pointer to the source vector or vector array.
srcStride0	Stride between the vectors in the source array.
srcStride2	Stride between the elements in the source vector(s).
srcRoiShift	ROI shift in the source vector.
val	The constant.
pVal, *ppVal*	Pointer to the source array of constants.
valStride0	Stride between the constants in the source array of constants.
valRoiShift	ROI shift for the source array of constants.
pDst, *ppDst*	Pointer to the destination vector or vector array.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector.
dstRoiShift	ROI shift in the destination vector.
len	Vector length.
count	The number of vectors (constants) in the array.

Description

The function `ippmMul` is declared in the `ippm.h` header file. The function multiplies the elements of the source vector by a constant and stores the result in the destination vector:

$$dst[i] = val \times src[i],\ 0 \le i < len.$$

To clarify how the function operates on arrays of vectors and constants, see the "Operations with arrays of objects" section in Chapter 2.

Examples of calling the function `ippmMul` operating with arrays of constants are similar to those of the function `ippmAdd` (see Example "ippmAdd_vaca_32f", Example "ippmAdd_vaca_32f_L", and Example "ippmAdd_vca_32f_P").

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

Example

The code example below demonstrates how to use the function `ippmMul_vac_32f`.

```
IppStatus mul_vac_32f(void) {
    /* Source data */
    Ipp32f pSrc[2*6] = { 2, 1, 3, 1, 4, 1,
                         5, 1, 6, 1, 7, 1 };
    /*
    // Elements of interest: 2 vectors with length=3
    */
    int srcStride2 = 2*sizeof(Ipp32f);
    int srcStride0 = 6*sizeof(Ipp32f);
    Ipp32f pDst[6*2] = {0};
    int dstStride2 = 2*sizeof(Ipp32f);
    int dstStride0 = 6*sizeof(Ipp32f);

    Ipp32f val=2.0;
    int length = 3;
    int count  = 2;

    IppStatus status = ippmMul_vac_32f((const Ipp32f*) pSrc, srcStride0,
        srcStride2, val, pDst, dstStride0, dstStride2, length, count);

    /*
    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */
    if (status == ippStsNoErr) {
        printf_va_Ipp32f("2 source vectors:", pSrc, 6, 2, status);
        printf_va_Ipp32f("2 destination vectors:", pDst, 6, 2, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
```

```
    return status;
}
```

Output:

```
2 source vectors:
2.000000  1.000000  3.000000  1.000000  4.000000  1.000000
5.000000  1.000000  6.000000  1.000000  7.000000  1.000000
2 destination vectors:
4.000000  0.000000  6.000000 0.000000  8.000000  0.000000
10.000000  0.000000  12.000000  0.000000  14.000000  0.000000
```

CrossProduct

Computes cross product of two 3D vectors.

Syntax

Case 1: Vector - vector operation

```
IppStatus ippmCrossProduct_vv_32f(const Ipp32f* pSrc1, int src1Stride2, const Ipp32f*
pSrc2, int src2Stride2, Ipp32f* pDst, int dstStride2);
```

```
IppStatus ippmCrossProduct_vv_64f(const Ipp64f* pSrc1, int src1Stride2, const Ipp64f*
pSrc2, int src2Stride2, Ipp64f* pDst, int dstStride2);
```

```
IppStatus ippmCrossProduct_vv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, const Ipp32f**
ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift);
```

```
IppStatus ippmCrossProduct_vv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, const Ipp64f**
ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift);
```

Case 2: Vector - vector array operation

```
IppStatus ippmCrossProduct_vva_32f(const Ipp32f* pSrc1, int src1Stride2, const Ipp32f*
pSrc2, int src2Stride0, int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride2,
int count);
```

```
IppStatus ippmCrossProduct_vva_64f(const Ipp64f* pSrc1, int src1Stride2, const Ipp64f*
pSrc2, int src2Stride0, int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride2,
int count);
```

```
IppStatus ippmCrossProduct_vva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, const Ipp32f**
ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst, int dstRoiShift, int dstStride0,
int count);
```

```
IppStatus ippmCrossProduct_vva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, const Ipp64f**
ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst, int dstRoiShift, int dstStride0,
int count);
```

```
IppStatus ippmCrossProduct_vva_32f_L(const Ipp32f* pSrc1, int src1Stride2, const Ipp32f**
ppSrc2, int src2RoiShift, int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2,
int count);
```

```
IppStatus ippmCrossProduct_vva_64f_L(const Ipp64f* pSrc1, int src1Stride2, const Ipp64f**
ppSrc2, int src2RoiShift, int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2,
int count);
```

Case 3: Vector array - vector operation

```
IppStatus ippmCrossProduct_vav_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride2,
const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride2, int
count);
```

```
IppStatus ippmCrossProduct_vav_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride2,
const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride2, int
count);
```

```
IppStatus ippmCrossProduct_vav_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift,
int dstStride0, int count);
```

```
IppStatus ippmCrossProduct_vav_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift,
int dstStride0, int count);
```

```
IppStatus ippmCrossProduct_vav_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride2, const Ipp32f* pSrc2, int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int
dstStride2, int count);
```

```
IppStatus ippmCrossProduct_vav_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride2, const Ipp64f* pSrc2, int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int
dstStride2, int count);
```

Case 4: Vector array - vector array operation

```
IppStatus ippmCrossProduct_vava_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride2,
const Ipp32f* pSrc2, int src2Stride0, int src2Stride2, Ipp32f* pDst, int dstStride0, int
dstStride2, int count);
```

```
IppStatus ippmCrossProduct_vava_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride2,
const Ipp64f* pSrc2, int src2Stride0, int src2Stride2, Ipp64f* pDst, int dstStride0, int
dstStride2, int count);
```

```
IppStatus ippmCrossProduct_vava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** pDst, int
dstRoiShift, int dstStride0, int count);
```

```
IppStatus ippmCrossProduct_vava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst,
int dstRoiShift, int dstStride0, int count);
```

```
IppStatus ippmCrossProduct_vava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp32f** ppDst,
int dstRoiShift, int dstStride2, int count);
```

```
IppStatus ippmCrossProduct_vava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp64f** ppDst,
int dstRoiShift, int dstStride2, int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the first source vector or vector array.
src1Stride0	Stride between the vectors in the first source vector array.
src1Stride2	Stride between the elements in the first source vector(s).
src1RoiShift	ROI shift in the first source vector.
pSrc2, ppSrc2	Pointer to the second source vector or vector array.
src2Stride0	Stride between the vectors in the second source vector array.
src2Stride2	Stride between the elements in the second source vector(s).
src2RoiShift	ROI shift in the second source vector(s).
pDst, ppDst	Pointer to the destination vector or vector array.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector.
dstRoiShift	ROI shift in the destination vector.
count	Number of vectors in the array.

Description

The function `ippmCrossProduct` is declared in the `ippm.h` header file. The function composes a vector product of two source vectors. The first element in the destination vector is obtained by subtracting the product of the third element in the first source vector and the second element in the second source vector from the product of the second element in the first source vector and the third element in the second source vector.

The following relations are used in computing the first destination element and the other two elements:

$$dst[0] = src1[1] \times src2[2] - src1[2] \times src2[1]$$

$$dst[1] = src1[2] \times src2[0] - src1[0] \times src2[2]$$

$$dst[2] = src1[0] \times src2[1] - src1[1] \times src2[0].$$

The result is stored in the destination vector.

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.

`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

NOTE. The function `ippmCrossProduct` is defined only for three-dimensional vectors and fails with all other argument types.

DotProduct

Computes dot product of two vectors.

Syntax

Case 1: Vector - vector operation

```
IppStatus ippmDotProduct_vv_32f(const Ipp32f* pSrc1, int src1Stride2, const Ipp32f* pSrc2,
int src2Stride2, Ipp32f* pDst, int len);
```

```
IppStatus ippmDotProduct_vv_64f(const Ipp64f* pSrc1, int src1Stride2, const Ipp64f* pSrc2,
int src2Stride2, Ipp64f* pDst, int len);
```

```
IppStatus ippmDotProduct_vv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, const Ipp32f**
ppSrc2, int src2RoiShift, Ipp32f* pDst, int len);
```

```
IppStatus ippmDotProduct_vv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, const Ipp64f**
ppSrc2, int src2RoiShift, Ipp64f* pDst, int len);
```

Case 2: Vector array - vector operation

```
IppStatus ippmDotProduct_vav_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride2,
const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst, int len, int count);
```

```
IppStatus ippmDotProduct_vav_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride2,
const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst, int len, int count);
```

```
IppStatus ippmDotProduct_vav_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f* pDst, int len, int count);
```

```
IppStatus ippmDotProduct_vav_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f* pDst, int len, int count);
```

```
IppStatus ippmDotProduct_vav_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride2,
const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst, int len, int count);
```

```
IppStatus ippmDotProduct_vav_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride2,
const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst, int len, int count);
```

Case 3: Vector array - vector array operation

```
IppStatus ippmDotProduct_vava_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride2,
const Ipp32f* pSrc2, int src2Stride0, int src2Stride2, Ipp32f* pDst, int len, int count);
```

```
IppStatus ippmDotProduct_vava_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride2,
const Ipp64f* pSrc2, int src2Stride0, int src2Stride2, Ipp64f* pDst, int len, int count);
```

```
IppStatus ippmDotProduct_vava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f* pDst, int
len, int count);
```

```
IppStatus ippmDotProduct_vava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f* pDst, int
len, int count);
```

```
IppStatus ippmDotProduct_vava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp32f* pDst, int
len, int count);
```

```
IppStatus ippmDotProduct_vava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp64f* pDst, int
len, int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the first source vector or vector array.
src1Stride0	Stride between the vectors in the first source vector array.
src1Stride2	Stride between the elements in the first source vector(s).
src1RoiShift	ROI shift in the first source vector.
pSrc2, ppSrc2	Pointer to the second source vector or vector array.
src2Stride0	Stride between the vectors in the second source vector array.
src2Stride2	Stride between the elements in the second source vector(s).
src2RoiShift	ROI shift in the second source vector(s).
pDst	Pointer to the destination value or array of values.
len	Vector length.
count	Number of vectors in the array.

Description

The function `ippmDotProduct` is declared in the `ippm.h` header file. The function computes the inner (dot) product of two source vectors by adding together the products of their respective elements and storing the resulting value in *pDst*:

$$dst = \sum_{i} src1[i] \times src2[i],\ 0 \le i < len.$$

If the operation is performed on a vector array, the resulting values are stored sequentially in the array with the pointer *pDst*.

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is `NULL`.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.

`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

Example

The code example below demonstrates how to use the function `ippmDotProduct_vav_32f`.

```
IppStatus dotProduct_vav_32f(void) {
    /* Src1 is 3 vectors with length=4 */
    Ipp32f pSrc1[3*4] = { 1,  2,  4,  8,
                          11, 22, 44, 88,
                          22, 44, 88, 176 };

    /* Src2 is vector with length=4 */
    Ipp32f pSrc2[4] = { 1, 0.5, 0.25 , 0.125 };
    /* Standard description for source vectors */
    int src1Stride2 = sizeof(Ipp32f);
    int src1Stride0 = 4*sizeof(Ipp32f);
    int src2Stride2 = sizeof(Ipp32f);



    int length = 4;
    int count  = 3;

    Ipp32f pDst[3];  /* Destination is array of values */

    IppStatus status = ippmDotProduct_vav_32f((const Ipp32f*)pSrc1,
        src1Stride0, src1Stride2, (const Ipp32f*)pSrc2, src2Stride2,
        pDst, length, count);

    /*
    // It is recommended to check return status

    // to detect wrong input parameters, if any
```

```
    */
    if (status == ippStsNoErr) {
        printf_va_Ipp32f("Src1 is 3 vectors:", pSrc1, 4, 3, status);
        printf_va_Ipp32f("Src2 is vector:", pSrc2, 4, 1, status);
        printf_va_Ipp32f("Dst is 3 values:", pDst, 3, 1, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }


    return status;
}
```

Output:

```
Src1 is 3 vectors:
1.000000  2.000000  4.000000  8.000000
11.000000  22.000000  44.000000  88.000000
22.000000  44.000000  88.000000  176.000000
Src2 is vector:
1.000000  0.500000  0.250000  0.125000
Dst is 3 values:
4.000000  44.000000  88.000000
```

L2Norm

Computes vector L2 norm.

Syntax

Case 1: Vector operation

IppStatus ippmL2Norm_v_32f(const Ipp32f* *pSrc*, int *srcStride2*, Ipp32f* *pDst*, int *len*);

IppStatus ippmL2Norm_v_64f(const Ipp64f* *pSrs*, int *srcStride2*, Ipp64f* *pDst*, int *len*);

IppStatus ippmL2Norm_v_32f_P(const Ipp32f** *ppSrc*, int *srcRoiShift*, Ipp32f* *pDst*, int *len*);

IppStatus ippmL2Norm_v_64f_P(const Ipp64f** *ppSrc*, int *srcRoiShift*, Ipp64f* *pDst*, int *len*);

Case 2: Vector array operation

IppStatus ippmL2Norm_va_32f(const Ipp32f* *pSrc*, int *srcStride0*, int *srcStride2*, Ipp32f* *pDst*, int *len*, int *count*);

```
IppStatus ippmL2Norm_va_64f(const Ipp64f* pSrc, int srcStride0, int srcStride2, Ipp64f*
pDst, int len, int count);
```

```
IppStatus ippmL2Norm_va_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0,
Ipp32f* pDst, int len, int count);
```

```
IppStatus ippmL2Norm_va_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0,
Ipp64f* pDst, int len, int count);
```

```
IppStatus ippmL2Norm_va_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride2,
Ipp32f* pDst, int len, int count);
```

```
IppStatus ippmL2Norm_va_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride2,
Ipp64f* pDst, int len, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source vector or vector array.
srcStride0	Stride between the vectors in the source vector array.
srcStride2	Stride between the elements in the source vector(s).
srcRoiShift	ROI shift in the source vector(s).
pDst	Pointer to the destination value or array of values.
len	Vector length.
count	Number of vectors in the array.

Description

The function `ippmL2Norm` is declared in the `ippm.h` header file. The function calculates L2 norm of the source vector and stores the result in *pDst*. The destination value is the square root of the sum of all the squared elements in the source vector, or

$$dst = \sqrt{\sum_i src[i]^2},\ 0 \le i < len.$$

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is `NULL`.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

Example

The code example below demonstrates how to use the function `ippmL2Norm_va_32f_P`.

```
IppStatus l2norm_va_32f_P(void) {
    /* Source data */
    Ipp32f src[3*4] = { 1.1f, 2.1f, 3.1f, 4.1f,
                        1.2f, 2.2f, 3.2f, 4.2f,
                        1.3f, 2.3f, 3.3f, 4.3f };

    /*
    // The first operand is 4 vector-columns with length=3
    // Pointer description:
    // ppSrc1[0] pointer to the first element of the first column
    // ppSrc1[1] pointer to the second element of the first column
    // ppSrc1[2] pointer to the third element  of the first column
    */
    Ipp32f* ppSrc[3] = { src, src+4, src+8 };
    int srcRoiShift = 0;
    int srcStride0 = sizeof(Ipp32f); /* Stride between columns */
    int length = 3;
    /*
    // Destination is 4 values
    */
    Ipp32f pDst[4];
    int count = 4;
    IppStatus status = ippmL2Norm_va_32f_P((const Ipp32f**)ppSrc,
       srcRoiShift, srcStride0, pDst, length, count);

    /*
    // It is recommended to check return status
    // to detect wrong input parameters, if any
```

```
    */
    if (status == ippStsNoErr) {
        printf_va_Ipp32f("Dst is 4 values:", pDst, 4, 1, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
    return status;
}
```

Output:

```
Dst is 4 values:
2.083267  3.813135  5.544366  7.275988
```

LComb

Composes linear combination of two vectors.

Syntax

Case 1: Vector - vector operation

IppStatus ippmLComb_vv_32f(const Ipp32f* *pSrc1*, int *src1Stride2*, Ipp32f *scale1*, const Ipp32f* *pSrc2*, int *src2Stride2*, Ipp32f *scale2*, Ipp32f* *pDst*, int *dstStride2*, int *len*);

IppStatus ippmLComb_vv_64f(const Ipp64f* *pSrc1*, int *src1Stride2*, Ipp64f *scale1*, const Ipp64f* *pSrc2*, int *src2Stride2*, Ipp64f *scale2*, Ipp64f* *pDst*, int *dstStride2*, int *len*);

IppStatus ippmLComb_vv_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, Ipp32f *scale1*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, Ipp32f *scale2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *len*);

IppStatus ippmLComb_vv_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, Ipp64f *scale1*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, Ipp64f *scale2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *len*);

Case 2: Vector array - vector operation

IppStatus ippmLComb_vav_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride2*, Ipp32f *scale1*, const Ipp32f* *pSrc2*, int *src2Stride2*, Ipp32f *scale2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmLComb_vav_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride2*, Ipp64f *scale1*, const Ipp64f* *pSrc2*, int *src2Stride2*, Ipp64f *scale2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *len*, int *count*);

IppStatus ippmLComb_vav_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, Ipp32f *scale1*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, Ipp32f *scale2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *len*, int *count*);

```
IppStatus ippmLComb_vav_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
Ipp64f scale1, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f scale2, Ipp64f** ppDst,
int dstRoiShift, int dstStride0, int len, int count);
```

```
IppStatus ippmLComb_vav_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride2,
Ipp32f scale1, const Ipp32f* pSrc2, int src2Stride2, Ipp32f scale2, Ipp32f** ppDst, int
dstRoiShift, int dstStride2, int len, int count);
```

```
IppStatus ippmLComb_vav_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride2,
Ipp64f scale1, const Ipp64f* pSrc2, int src2Stride2, Ipp64f scale2, Ipp64f** ppDst, int
dstRoiShift, int dstStride2, int len, int count);
```

Case 3: Vector array - vector array operation

```
IppStatus ippmLComb_vava_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride2, Ipp32f
scale1, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2, Ipp32f scale2, Ipp32f*
pDst, int dstStride0, int dstStride2, int len, int count);
```

```
IppStatus ippmLComb_vava_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride2, Ipp64f
scale1, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2, Ipp64f scale2, Ipp64f*
pDst, int dstStride0, int dstStride2, int len, int count);
```

```
IppStatus ippmLComb_vava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
Ipp32f scale1, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f scale2,
Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len, int count);
```

```
IppStatus ippmLComb_vava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
Ipp64f scale1, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f scale2,
Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int count);
```

```
IppStatus ippmLComb_vava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride2,
Ipp32f scale1, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp32f scale2,
Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

```
IppStatus ippmLComb_vava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride2,
Ipp64f scale1, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp64f scale2,
Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the first source vector or vector array.
src1Stride0	Stride between the vectors in the first source vector array.
src1Stride2	Stride between the elements in the first source vector(s).
src1RoiShift	ROI shift in the first source vector.
pSrc2, ppSrc2	Pointer to the second source vector or vector array.
src2Stride0	Stride between the vectors in the second source vector array.
src2Stride2	Stride between the elements in the second source vector(s).
src2RoiShift	ROI shift in the second source vector(s).
pDst, ppDst	Pointer to the destination vector or vector array.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector.

dstRoiShift	ROI shift in the destination vector.
scale1	First multiplier.
scale2	Second multiplier.
len	Vector length.
count	Number of vectors in the array.

Description

The function `ippmLComb` is declared in the `ippm.h` header file. The function composes a linear combination of two vectors by multiplying the first source vector by *scale1*, adding the result to the second source vector multiplied by *scale2*, and storing the resulting vector in *pDst*.

$$dst[i] = scale1 \times src1[i] + scale2 \times src2[i],$$

$$0 \le i < len.$$

The following example demonstrates how to use the function `ippmLComb_vava_32f`. For more information, see also examples in the Getting Started chapter.

Example 4-11 ippmLComb_vava_32f

```
IppStatus lcomb_vava_32f(void) {
    /* Src1 data: 4 vectors with length=3, Stride2=2*sizeof(Ipp32f) */
    Ipp32f pSrc1[4*6] = { 1, 0,  2,  0, 3,  0,
                          4, 0,  5,  0, 6,  0,
                          7, 0,  8,  0, 9,  0,
                          6, 0,  4,  0, 2, 0 };

    int src1Stride2 = 2*sizeof(Ipp32f);
    int src1Stride0 = 6*sizeof(Ipp32f);

   /* Src2 data: 4 vectors with length=3, Stride2=sizeof(Ipp32f) */
    Ipp32f pSrc2[4*3] = { -3, -1, -2,
                          -7, -3, -4,
                          -4, -5, -6,
                          -5, -2, -3 };

    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride0 = 3*sizeof(Ipp32f);

    /* Dst is 4 vectors with length=3, Stride2=sizeof(Ipp32f) */
    Ipp32f pDst[4*3];
    int dstStride2=sizeof(Ipp32f);
    int dstStride0=3*sizeof(Ipp32f);

    Ipp32f scale1 = 0.5;
    Ipp32f scale2 = 1.5;
    int length = 3;
    int count  = 4;

    IppStatus status = ippmLComb_vava_32f((const Ipp32f*)pSrc1,
        src1Stride0, src1Stride2, scale1,
        (const Ipp32f*)pSrc2, src2Stride0, src2Stride2, scale2,
        pDst, dstStride0, dstStride2, length, count);
```

```
    /*
    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */
    if (status == ippStsNoErr) {
        printf_va_Ipp32f("4 destination vectors:", pDst, 3, 4, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
    return status;
}
```

The program above produces the following output:

```
4 destination vectors:
-4.000000  -0.500000  -1.500000
-8.500000  -2.000000  -3.000000
-2.500000  -3.500000  -4.500000
-4.500000  -1.000000  -3.500000
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

Matrix Algebra Functions

5

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) functions for small matrices that perform matrix algebra operations.

Matrix Algebra functions

Function Base Name	Operation
`Transpose`	Performs matrix transposition.
`Invert`	Computes matrix inverse.
`FrobNorm`	Computes matrix Frobenius norm.
`Det`	Computes matrix determinant.
`Trace`	Computes matrix trace.
`Mul`	Multiplies matrix by a constant, vector, or another matrix.
`Add`	Adds matrix to another matrix.
`Sub`	Subtracts matrix from another matrix.
`Gaxpy`	Performs the "gaxpy" operation on a matrix.
`AffineTransform3DH`	Performs an arbitrary affine transformation with an array of 3D vectors in the Homogeneous coordinate space.

Transpose

Performs matrix transposition.

Syntax

Case 1: Matrix operation

```
IppStatus ippmTranspose_m_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2, int width, int height, Ipp32f* pDst, int dstStride1, int dstStride2);
```

```
IppStatus ippmTranspose_m_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2, int width, int height, Ipp64f* pDst, int dstStride1, int dstStride2);
```

```
IppStatus ippmTranspose_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int width, int height, Ipp32f** ppDst, int dstRoiShift);
```

```
IppStatus ippmTranspose_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int width, int height, Ipp64f** ppDst, int dstRoiShift);
```

Case 2: Matrix array operation

```
IppStatus ippmTranspose_ma_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1, int srcStride2, int width, int height, Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int count);
```

```
IppStatus ippmTranspose_ma_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1, int
srcStride2, int width, int height, Ipp64f* pDst, int dstStride0, int dstStride1, int
dstStride2, int count);
```

```
IppStatus ippmTranspose_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0,
int width, int height, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int count);
```

```
IppStatus ippmTranspose_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0,
int width, int height, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int count);
```

```
IppStatus ippmTranspose_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1,
int srcStride2, int width, int height, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
int dstStride2, int count);
```

```
IppStatus ippmTranspose_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1,
int srcStride2, int width, int height, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
int dstStride2, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
width	Source matrix width.
height	Source matrix height.
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
count	Number of matrices in the array.

Description

The function `ippmTranspose` is declared in the `ippm.h` header file. The function transposes the source matrix and stores the result in *pDst* or *ppDst*. The destination is obtained from the source matrix by transforming the columns of the source to the rows in the destination for each matrix element:

dst[j][i] = src[i][j],

$0 \le i < height$, $0 \le j < width$.

Note that the destination matrix must have the number of columns equal to *height* and the number of rows equal to *width*.

Return Values

ippStsOk	Returns no error.

`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by the size of the data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the *roiShift* value is negative or not divisible by the size of the data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

Example

The code example below demonstrates how to use the function `ippmTranspose_m_32f`.

```
IppStatus transpose_m_32f(void) {
    /* Source matrix with width=4 and height=3 */
    Ipp32f pSrc[3*4] = { 1,  2, 3, 4,
                         5,  6, 7, 8,
                         9,  0, 1, 2 };

    /* Destination matrix with width=3 and height=4 */
    Ipp32f pDst[4*3];
    /* Standard description for source and destination matrices */
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 4*sizeof(Ipp32f);
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);

    IppStatus status = ippmTranspose_m_32f((const Ipp32f*)pSrc,
        srcStride1, srcStride2, 4, 3, pDst, dstStride1, dstStride2);

    /*

    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */

    if (status == ippStsNoErr) {

        printf_m_Ipp32f("Transposed matrix:", pDst, 3, 4, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }

    return status;
}
```

Output:

Transposed matrix:

```
1.000000  5.000000  9.000000

2.000000  6.000000  0.000000

3.000000  7.000000  1.000000
```

```
4.000000  8.000000  2.000000
```

The code example below demonstrates how to use the function `ippmTranspose_m_32f_P`.

```
IppStatus transpose_m_32f_P(void) {
    /* Source data */
    Ipp32f src[2*6] = { 1, 0, 0, 2, 0, 3,
                        0, 4, 5, 0, 0, 6 };
    /*
    // Nonzero elements of interest are referred by mask using
    // Poiner descriptor:
    // Src width=3, height=2
    */ int srcRoiShift = 0;
    Ipp32f* ppSrc[2*3] = { src,   src+3, src+5,
                           src+7, src+8, src+11 };

    /*
    // Pointer description for destination matrix:
    // Dst width=2, height=3
    */

    Ipp32f  dst[3*2];
    int dstRoiShift = 0;
    Ipp32f* ppDst[3*2] = { dst,   dst+1,
                           dst+2, dst+3,
                           dst+4, dst+5 };

    IppStatus status = ippmTranspose_m_32f_P((const Ipp32f**)ppSrc,
        srcRoiShift, 3, 2, ppDst, dstRoiShift);

    /*

    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */

    if (status == ippStsNoErr) {

        printf_m_Ipp32f_P("Transposed matrix:", ppDst, 2, 3, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }


   /*

    return status;
}
```

Output:

```
Transposed matrix:

1.000000 4.000000

2.000000 5.000000

3.000000 6.000000
```

The code example below demonstrates how to use the function `ippmTranspose_ma_32f_L`.

```
IppStatus transpose_ma_32f_L(void) {
    /* Source data:
    // 3 matrices with width=4 and height=2
    */
    Ipp32f src_a[2*4] = { 10, 11, 12, 13, 24, 25, 26, 27 };
    Ipp32f src_b[2*4] = { 30, 31, 32, 33, 44, 45, 46, 47 };
    Ipp32f src_c[2*4] = { 50, 51, 52, 53, 64, 65, 66, 67 };
    /*
    // Layout description for 3 source matrices
    */
    int srcRoiShift = 0;
    int srcStride2  = sizeof(Ipp32f);
    int srcStride1  = 4*sizeof(Ipp32f);

    Ipp32f* ppSrc[3] = { src_a, src_b, src_c };
    /*
    // Layout description for destination matrices:
    // width=2, height=4, count=3
    */
    Ipp32f dst_a[4*2];
    Ipp32f dst_b[4*2];
    Ipp32f dst_c[4*2];

    Ipp32f* ppDst[3] = { dst_a, dst_b, dst_c };



    int dstRoiShift = 0;
    int dstStride2  = sizeof(Ipp32f);
    int dstStride1  = 2*sizeof(Ipp32f);

    IppStatus status = ippmTranspose_ma_32f_L((const Ipp32f**)ppSrc,
        srcRoiShift, srcStride1, srcStride2, 4, 2,
        ppDst, dstRoiShift, dstStride1, dstStride2, 3);

    /*

    // It is recommended to check return status


    // to detect wrong input parameters, if any


    */

    if (status == ippStsNoErr) {

        printf_ma_Ipp32f_L("3 transposed matrices:", ppDst,2,4,3, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }


    return status;
}
```

Output:

3 transposed matrices:

```
10.000000 24.000000     30.000000 44.000000     50.000000 64.000000
11.000000 25.000000     31.000000 45.000000     51.000000 65.000000
12.000000 26.000000     32.000000 46.000000     52.000000 66.000000
13.000000 27.000000     33.000000 47.000000     53.000000 67.000000
```

Invert

Computes matrix inverse.

Syntax

Case 1: Matrix operation

```
IppStatus ippmInvert_m_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2, Ipp32f* pBuffer, Ipp32f* pDst, int dstStride1, int dstStride2, int widthHeight);
```

```
IppStatus ippmInvert_m_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2, Ipp64f* pBuffer, Ipp64f* pDst, int dstStride1, int dstStride2, int widthHeight);
```

```
IppStatus ippmInvert_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f* pBuffer, Ipp32f** ppDst, int dstRoiShift, int widthHeight);
```

```
IppStatus ippmInvert_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f* pBuffer, Ipp64f** ppDst, int dstRoiShift, int widthHeight);
```

Case 2: Matrix array operation

```
IppStatus ippmInvert_ma_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1, int srcStride2, Ipp32f* pBuffer, Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmInvert_ma_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1, int srcStride2, Ipp64f* pBuffer, Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmInvert_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, Ipp32f* pBuffer, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int widthHeight, int count);
```

```
IppStatus ippmInvert_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0, Ipp64f* pBuffer, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int widthHeight, int count);
```

```
IppStatus ippmInvert_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1, int srcStride2, Ipp32f* pBuffer, Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmInvert_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1, int srcStride2, Ipp64f* pBuffer, Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int widthHeight, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.

srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
widthHeight	Size of the square matrix.
pBuffer	Pointer to a pre-allocated buffer to be used for internal computations. Size of the buffer must be at least $widthHeight^2 + widthHeight$.
count	Number of matrices in the array.

Description

The function `ippmInvert` is declared in the `ippm.h` header file. The function finds the inverse of the source matrix and stores the result in *pDst* or *ppDst*. Upon completion,

$dst = src^{-1}$.

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsSingularErr	Returns an error when the source matrix is singular.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by the size of the data type.
ippStsRoiShiftMatrixErr	Returns an error when the *roiShift* value is negative or not divisible by the size of the data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

Example

The code example below demonstrates how to use the function `ippmInvert_m_32f`. All parameter settings and descriptor types are similar to those used in ippmTranspose. Refer to examples in the description of this function for details.

```
IppStatus invert_m_32f(void) {
   /* Source matrix with widthHeight=3 */
    Ipp32f pSrc[3*3] = { 1,  1, -1,
                        -1,  0,  2,
                        -1, -1,  2 };

    /*
    // Standard description for source and destination matrices
    */
```

```
    int widthHeight = 3;
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 3*sizeof(Ipp32f);

    Ipp32f pDst[3*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);

    Ipp32f pBuffer[3*3+3]; /* Buffer location */

    IppStatus status = ippmInvert_m_32f((const Ipp32f*)pSrc, srcStride1,
        srcStride2, pBuffer, pDst, dstStride1, dstStride2, 3);


    /*

    // It is required for Invert function to check return status


    // for catching wrong result in case of invalid input data


    */

    if (status == ippStsNoErr) {

        printf_m_Ipp32f("Inverted matrix:", pDst, 3, 3, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }


    return status;

}
```

Output:

Inverted matrix:

```
2.000000  -1.000000  2.000000

0.000000  1.000000  -1.000000

1.000000  0.000000  1.000000
```

FrobNorm

Computes matrix Frobenius.

Syntax

Case 1: Matrix operation

IppStatus ippmFrobNorm_m_32f(const Ipp32f* *pSrc*, int *srcStride1*, int *srcStride2*, int *width*, int *height*, Ipp32f* *pDst*);

IppStatus ippmFrobNorm_m_64f(const Ipp64f* *pSrc*, int *srcStride1*, int *srcStride2*, int *width*, int *height*, Ipp64f* *pDst*);

```
IppStatus ippmFrobNorm_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int width, int
height, Ipp32f* pDst);
```

```
IppStatus ippmFrobNorm_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int width, int
height, Ipp64f* pDst);
```

Case 2: Matrix array operation

```
IppStatus ippmFrobNorm_ma_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1, int
srcStride2, int width, int height, Ipp32f* pDst, int count);
```

```
IppStatus ippmFrobNorm_ma_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1, int
srcStride2, int width, int height, Ipp64f* pDst, int count);
```

```
IppStatus ippmFrobNorm_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0,
int width, int height, Ipp32f* pDst, int count);
```

```
IppStatus ippmFrobNorm_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0,
int width, int height, Ipp64f* pDst, int count);
```

```
IppStatus ippmFrobNorm_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1,
int srcStride2, int width, int height, Ipp32f* pDst, int count);
```

```
IppStatus ippmFrobNorm_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1,
int srcStride2, int width, int height, Ipp64f* pDst, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDst	Pointer to the destination value or array of values.
width	Matrix width.
height	Matrix height.
count	Number of matrices in the array.

Description

The function `ippmFrobNorm` is declared in the `ippm.h` header file. The function calculates Frobenius norm of the source matrix and stores the result in *pDst*. The destination value is the square root of the sum of all the squared elements in the source matrix, or

$$dst = \sqrt{\sum_{i,j} src[i][j]^2}, 0 \le i < height, 0 \le j < width.$$

If the operation is performed on a vector array, the resulting values are stored sequentially in the array with the pointer *pDst*.

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by the size of the data type.
ippStsRoiShiftMatrixErr	Returns an error when the *roiShift* value is negative or not divisible by the size of the data type.
ippStsCountMatrixErr	Returns an error when the *count* value is less or equal to zero.

Example

The code example below demonstrates how to use the function ippmFrobNorm_ma_32f_L.

```
IppStatus frobnorm_ma_32f_L(void) {
   /*  Source data  */
    Ipp32f a[2*3] = { 1.0f, 1.1f, 1.2f, 1.3f, 1.4f, 1.5f };
    Ipp32f b[2*3] = { 2.0f, 2.1f, 2.2f, 2.3f, 2.4f, 2.5f };
    Ipp32f c[2*3] = { 3.0f, 3.1f, 3.2f, 3.3f, 3.4f, 3.5f };
    Ipp32f d[2*3] = { 4.0f, 4.1f, 4.2f, 4.3f, 4.4f, 4.5f };
    /*
    // Layout description for 4 source matrices:
    */
    int width  = 3;
    int height = 2;
    int count  = 4;
    Ipp32f* ppSrc[4] = { a, b, c, d };
    int srcRoiShift = 0;
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 3*sizeof(Ipp32f);

    Ipp32f pDst[4];  /* Destination is array of values */

    IppStatus status = ippmFrobNorm_ma_32f_L((const Ipp32f**)ppSrc,
       srcRoiShift, srcStride1, srcStride2, width, height, pDst, count);

    /*

    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */

    if (status == ippStsNoErr) {

        printf_va_Ipp32f("Dst is 4 values:", pDst, 4, 1, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }

    return status;
}
```

Output:

Dst is 4 values:

```
3.090307  5.527205  7.971826  10.418734
```

Det

Computes matrix determinant.

Syntax

Case 1: Matrix operation

```
IppStatus ippmDet_m_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2, int widthHeight,
Ipp32f* pBuffer, Ipp32f* pDst);
```

```
IppStatus ippmDet_m_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2, int widthHeight,
Ipp64f* pBuffer, Ipp64f* pDst);
```

```
IppStatus ippmDet_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int widthHeight, Ipp32f*
pBuffer, Ipp32f* pDst);
```

```
IppStatus ippmDet_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int widthHeight, Ipp64f*
pBuffer, Ipp64f* pDst);
```

Case 2: Matrix array operation

```
IppStatus ippmDet_ma_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1, int srcStride2,
int widthHeight, Ipp32f* pBuffer, Ipp32f* pDst, int count);
```

```
IppStatus ippmDet_ma_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1, int srcStride2,
int widthHeight, Ipp64f* pBuffer, Ipp64f* pDst, int count);
```

```
IppStatus ippmDet_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, int
widthHeight, Ipp32f* pBuffer, Ipp32f* pDst, int count);
```

```
IppStatus ippmDet_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0, int
widthHeight, Ipp64f* pBuffer, Ipp64f* pDst, int count);
```

```
IppStatus ippmDet_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1, int
srcStride2, int widthHeight, Ipp32f* pBuffer, Ipp32f* pDst, int count);
```

```
IppStatus ippmDet_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1, int
srcStride2, int widthHeight, Ipp64f* pBuffer, Ipp64f* pDst, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDst	Pointer to the destination value or array of values.
widthHeight	Size of the square matrix.

pBuffer	Pointer to a pre-allocated buffer to be used for internal computations. Size of the buffer must be at least *widthHeight*^{2}+ *widthHeight*.
count	Number of matrices in the array.

Description

The function `ippmDet` is declared in the `ippm.h` header file. The function calculates the determinant of the source matrix and stores the result in *pDst*. Upon completion,

dst = det(src).

If the operation is performed on a vector array, the resulting values are stored sequentially in the array with the pointer *pDst*.

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by the size of the data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the *roiShift* value is negative or not divisible by the size of the data type.
`ippStsCountMatrixErr`	Returns an error when the *count* value is less or equal to zero.

Example

The code example below demonstrates how to use the function `ippmDet_ma_32f`.

```
IppStatus det_ma_32f(void) {
    /* Source data: 2 matrices with widthHeight=3 */
    Ipp32f pSrc[8*3] = { 5.0f, 1.1f, 1.2f,
                         1.3f, 1.4f, 1.5f,
                         2.0f, 2.1f, 2.2f,
                         0.0f, 0.0f, 0.0f,
                         6.3f, 2.4f, 2.5f,
                         3.0f, 3.1f, 3.2f,
                         4.3f, 4.4f, 4.5f,
                         0.0f, 0.0f, 0.0f };
    /* Standard description for 2 source matrices */
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 3*sizeof(Ipp32f);
    int srcStride0 = 4*3*sizeof(Ipp32f);

    int widthHeight = 3;
    int count = 2;

    Ipp32f pDst[2];         /* Destination is array of values */
    Ipp32f pBuffer[3*3+3];  /* Buffer location */
```

```
    IppStatus status = ippmDet_ma_32f((const Ipp32f*)pSrc, srcStride0,
        srcStride1, srcStride2, widthHeight, pBuffer, pDst, count);
    /*
    // It is recommended to check return status
    // to detect wrong input parameters, if any
    */
    if (status == ippStsNoErr) {
        printf_va_Ipp32f("Dst is 2 values:", pDst, 2, 1, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }

    return status;
}
```

Output:

```
Dst is 2 values:
-0.279999  -0.520004
```

Trace

Computes matrix trace.

Syntax

Case 1: Matrix operation

`IppStatus ippmTrace_m_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2, int widthHeight, Ipp32f* pDst);`

`IppStatus ippmTrace_m_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2, int widthHeight, Ipp64f* pDst);`

`IppStatus ippmTrace_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int widthHeight, Ipp32f* pDst);`

`IppStatus ippmTrace_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int widthHeight, Ipp64f* pDst);`

Case 2: Matrix array operation

`IppStatus ippmTrace_ma_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1, int srcStride2, int widthHeight, Ipp32f* pDst, int count);`

`IppStatus ippmTrace_ma_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1, int srcStride2, int widthHeight, Ipp64f* pDst, int count);`

`IppStatus ippmTrace_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, int widthHeight, Ipp32f* pDst, int count);`

```
IppStatus ippmTrace_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0, int widthHeight, Ipp64f* pDst, int count);
```

```
IppStatus ippmTrace_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1, int srcStride2, int widthHeight, Ipp32f* pDst, int count);
```

```
IppStatus ippmTrace_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1, int srcStride2, int widthHeight, Ipp64f* pDst, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDst	Pointer to the destination value or array of values.
widthHeight	Size of the square matrix.
count	Number of matrices in the array.

Description

The function `ippmTrace` is declared in the `ippm.h` header file. The function sums up the elements along the principal diagonal of the source matrix and stores the result in *pDst*. The following formula illustrates how trace is calculated:

$$dst = \sum_i src[i][i], 0 \le i < widthHeight.$$

If the operation is performed on a vector array, the resulting values are stored sequentially in the array with the pointer *pDst*.

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by the size of the data type.
ippStsRoiShiftMatrixErr	Returns an error when a *roiShift* value is negative or not divisible by the size of the data type.
ippStsCountMatrixErr	Returns an error when a *count* value is less or equal to zero.

Example

The code example demonstrates how to use the function `ippmTrace_ma_32f_P`.

```
IppStatus trace_ma_32f_P(void) {
   /*
    // Source data:
    // 2 matrices with widthHeight=3
    */

    Ipp32f src[6*6] = { 1, 0, 0, 1, 0, 1,
                        0, 1, 1, 0, 0, 1,
                        0, 0, 1, 1, 0, 1,
                        2, 0, 0, 2, 0, 2,
                        0, 2, 2, 0, 0, 2,
                        0, 0, 2, 2, 0, 2};
    /*
    // Nonzero elements of interest are referred by mask using
    // pointer descriptor
    */
    int widthHeight=3;
    int count=2;
    int srcRoiShift= 0;
    int srcStride0 = 3*6*sizeof(Ipp32f);

    Ipp32f* ppSrc[3*3] = { src,    src+3,  src+5,
                           src+7,  src+8,  src+11,
                           src+14, src+15, src+17 };

    Ipp32f pDst[2]; /* Destination is array of values */

    IppStatus status = ippmTrace_ma_32f_P((const Ipp32f**)ppSrc,
        srcRoiShift, srcStride0, widthHeight, pDst, 2);

    /*

    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */

    if (status == ippStsNoErr) {

        printf_va_Ipp32f("Dst is 2 values:", pDst, 2, 1, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }

    return status;
}
```

Output:

```
Dst is 2 values:

3.000000  6.000000
```

Mul

Multiplies matrix by a constant, a vector or another matrix.

Syntax

Case 1: Matrix - constant operation

```
IppStatus ippmMul_mc_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2, Ipp32f val, Ipp32f* pDst, int dstStride1, int dstStride2, int width, int height);

IppStatus ippmMul_mc_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2, Ipp64f val, Ipp64f* pDst, int dstStride1, int dstStride2, int width, int height);

IppStatus ippmMul_mc_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int width, int height);

IppStatus ippmMul_mc_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int width, int height);
```

Case 2: Transposed matrix - constant operation

```
IppStatus ippmMul_tc_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2, Ipp32f val, Ipp32f* pDst, int dstStride1, int dstStride2, int width, int height);

IppStatus ippmMul_tc_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2, Ipp64f val, Ipp64f* pDst, int dstStride1, int dstStride2, int width, int height);

IppStatus ippmMul_tc_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int width, int height);

IppStatus ippmMul_tc_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f val, Ipp64f** pDst, int dstRoiShift, int width, int height);
```

Case 3: Matrix array - constant operation

```
IppStatus ippmMul_mac_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1, int srcStride2, Ipp32f val, Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmMul_mac_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1, int srcStride2, Ipp64f val, Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmMul_mac_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmMul_mac_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmMul_mac_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1, int srcStride2, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmMul_mac_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1, int srcStride2, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);
```

Case 4: Transposed matrix array - constant operation

```
IppStatus ippmMul_tac_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1, int
srcStride2, Ipp32f val, Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int
width, int height, int count);

IppStatus ippmMul_tac_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1, int
srcStride2, Ipp64f val, Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int
width, int height, int count);

IppStatus ippmMul_tac_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0, Ipp32f
val, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmMul_tac_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0, Ipp64f
val, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmMul_tac_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1, int
srcStride2, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2,
int width, int height, int count);

IppStatus ippmMul_tac_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1, int
srcStride2, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2,
int width, int height, int count);
```

Case 5: Matrix - vector operation

```
IppStatus ippmMul_mv_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride2, int src2Len, Ipp32f*
pDst, int dstStride2);

IppStatus ippmMul_mv_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride2, int src2Len, Ipp64f*
pDst, int dstStride2);

IppStatus ippmMul_mv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Width, int
src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Len, Ipp32f** ppDst, int
dstRoiShift);

IppStatus ippmMul_mv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Width, int
src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Len, Ipp64f** ppDst, int
dstRoiShift);
```

Case 6: Transposed matrix - vector operation

```
IppStatus ippmMul_tv_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride2, int src2Len, Ipp32f*
pDst, int dstStride2);

IppStatus ippmMul_tv_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride2, int src2Len, Ipp64f*
pDst, int dstStride2);

IppStatus ippmMul_tv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Width, int
src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Len, Ipp32f** ppDst, int
dstRoiShift);
```

```
IppStatus ippmMul_tv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Width, int
src1Height, const Ipp64f** pSrc2, int src2RoiShift, int src2Len, Ipp64f** ppDst, int
dstRoiShift);
```

Case 7: Matrix - vector array operation

```
IppStatus ippmMul_mva_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2, int
src2Len, Ipp32f* pDst, int dstStride0, int dstStride2, int count);

IppStatus ippmMul_mva_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2, int
src2Len, Ipp64f* pDst, int dstStride0, int dstStride2, int count);

IppStatus ippmMul_mva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Width, int
src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, int src2Len, Ipp32f**
ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Width, int
src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, int src2Len, Ipp64f**
ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mva_32f_L(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, int
src2Len, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int count);

IppStatus ippmMul_mva_64f_L(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, int
src2Len, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int count);
```

Case 8: Transposed matrix - vector array operation

```
IppStatus ippmMul_tva_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2, int
src2Len, Ipp32f* pDst, int dstStride0, int dstStride2, int count);

IppStatus ippmMul_tva_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2, int
src2Len, Ipp64f* pDst, int dstStride0, int dstStride2, int count);

IppStatus ippmMul_tva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Width, int
src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, int src2Len, Ipp32f**
ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Width, int
src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, int src2Len, Ipp64f**
ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tva_32f_L(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, int
src2Len, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int count);

IppStatus ippmMul_tva_64f_L(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, int
src2Len, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int count);
```

Case 9: Matrix array - vector operation

IIppStatus ippmMul_mav_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride2*, int *src2Len*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *count*);

IppStatus ippmMul_mav_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride2*, int *src2Len*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *count*);

IppStatus ippmMul_mav_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Len*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_mav_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Len*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_mav_32f_L(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride2*, int *src2Len*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *count*);

IppStatus ippmMul_mav_64f_L(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride2*, int *src2Len*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *count*);

Case 10: Transposed matrix array - vector operation

IppStatus ippmMul_tav_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride2*, int *src2Len*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *count*);

IppStatus ippmMul_tav_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride2*, int *src2Len*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *count*);

IppStatus ippmMul_tav_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Len*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_tav_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Len*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_tav_32f_L(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride2*, int *src2Len*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *count*);

IppStatus ippmMul_tav_64f_L(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride2*, int *src2Len*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *count*);

Case 11: Matrix array - vector array operation

IppStatus ippmMul_mava_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride0*, int *src2Stride2*, int *src2Len*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *count*);

IppStatus ippmMul_mava_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride0*, int *src2Stride2*, int *src2Len*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *count*);

IppStatus ippmMul_mava_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, int *src2Len*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_mava_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, int *src2Len*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_mava_32f_L(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride2*, int *src2Len*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *count*);

IppStatus ippmMul_mava_64f_L(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride2*, int *src2Len*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *count*);

Case 12: Transposed matrix array - vector array operation

IppStatus ippmMul_tava_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride0*, int *src2Stride2*, int *src2Len*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *count*);

IppStatus ippmMul_tava_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride0*, int *src2Stride2*, int *src2Len*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *count*);

IppStatus ippmMul_tava_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, int *src2Len*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_tava_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, int *src2Len*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_tava_32f_L(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride2*, int *src2Len*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *count*);

IppStatus ippmMul_tava_64f_L(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride2*, int *src2Len*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *count*);

Case 13: Matrix - matrix operation

IppStatus ippmMul_mm_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f* *pDst*, int *dstStride1*, int *dstStride2*);

IppStatus ippmMul_mm_64f(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp64f* *pDst*, int *dstStride1*, int *dstStride2*);

IppStatus ippmMul_mm_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp32f** *ppDst*, int *dstRoiShift*);

IppStatus ippmMul_mm_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp64f** *ppDst*, int *dstRoiShift*);

IppStatus ippmMul_mm4x4_32f(const Ipp32f** *pSrc1*, const Ipp32f** *pSrc2*, Ipp32f** *pDst*);

Case 14: Transposed matrix - matrix operation

IppStatus ippmMul_tm_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f* *pDst*, int *dstStride1*, int *dstStride2*);

IppStatus ippmMul_tm_64f(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp64f* *pDst*, int *dstStride1*, int *dstStride2*);

IppStatus ippmMul_tm_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp32f** *ppDst*, int *dstRoiShift*);

IppStatus ippmMul_tm_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp64f** *ppDst*, int *dstRoiShift*);

Case 15: Matrix - transposed matrix operation

IppStatus ippmMul_mt_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f* *pDst*, int *dstStride1*, int *dstStride2*);

IppStatus ippmMul_mt_64f(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp64f* *pDst*, int *dstStride1*, int *dstStride2*);

IppStatus ippmMul_mt_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp32f** *ppDst*, int *dstRoiShift*);

IppStatus ippmMul_mt_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp64f** *ppDst*, int *dstRoiShift*);

Case 16: Transposed matrix - transposed matrix operation

IppStatus ippmMul_tt_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f* *pDst*, int *dstStride1*, int *dstStride2*);

IppStatus ippmMul_tt_64f(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp64f* *pDst*, int *dstStride1*, int *dstStride2*);

IppStatus ippmMul_tt_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp32f** *ppDst*, int *dstRoiShift*);

IppStatus ippmMul_tt_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp64f** *ppDst*, int *dstRoiShift*);

Case 17: Matrix - matrix array operation

IppStatus ippmMul_mma_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride0*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_mma_64f(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride0*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_mma_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, int *src2Width*, int *src2Height*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_mma_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, int *src2Width*, int *src2Height*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_mma_32f_L(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_mma_64f_L(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *count*);

Case 18: Transposed matrix - matrix array operation

IppStatus ippmMul_tma_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride0*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *count*);

```
IppStatus ippmMul_tma_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int
src2Stride2, int src2Width, int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
int dstStride2, int count);
```

```
IppStatus ippmMul_tma_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Width, int
src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, int src2Width, int
src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int count);
```

```
IppStatus ippmMul_tma_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Width, int
src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, int src2Width, int
src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int count);
```

```
IppStatus ippmMul_tma_32f_L(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1, int
src2Stride2, int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
int dstStride2, int count);
```

```
IppStatus ippmMul_tma_64f_L(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1, int
src2Stride2, int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
int dstStride2, int count);
```

Case 19: Matrix - transposed matrix array operation

```
IppStatus ippmMul_mta_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride0, int src2Stride1, int
src2Stride2, int src2Width, int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1,
int dstStride2, int count);
```

```
IppStatus ippmMul_mta_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int
src2Stride2, int src2Width, int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
int dstStride2, int count);
```

```
IppStatus ippmMul_mta_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Width, int
src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, int src2Width, int
src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int count);
```

```
IppStatus ippmMul_mta_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Width, int
src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, int src2Width, int
src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int count);
```

```
IppStatus ippmMul_mta_32f_L(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1, int
src2Stride2, int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
int dstStride2, int count);
```

```
IppStatus ippmMul_mta_64f_L(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, int
src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1, int
src2Stride2, int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
int dstStride2, int count);
```

Case 20: Transposed matrix - transposed matrix array operation

IppStatus ippmMul_tta_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride0*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_tta_64f(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride0*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_tta_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, int *src2Width*, int *src2Height*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_tta_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, int *src2Width*, int *src2Height*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_tta_32f_L(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_tta_64f_L(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *count*);

Case 21: Matrix array - matrix operation

IppStatus ippmMul_mam_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_mam_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_mam_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_mam_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_mam_32f_L(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_mam_64f_L(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *count*);

Case 22: Transposed matrix array - matrix operation

IppStatus ippmMul_tam_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_tam_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_tam_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_tam_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_tam_32f_L(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_tam_64f_L(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *count*);

Case 23: Matrix array - transposed matrix operation

IppStatus ippmMul_mat_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_mat_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, int *src2Width*, int *src2Height*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *count*);

IppStatus ippmMul_mat_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmMul_mat_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Width*, int *src2Height*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

```
IppStatus ippmMul_mat_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1, int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int count);

IppStatus ippmMul_mat_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1, int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int count);
```

Case 24: Transposed matrix array - transposed matrix operation

```
IppStatus ippmMul_tat_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int count);

IppStatus ippmMul_tat_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int count);

IppStatus ippmMul_tat_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tat_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0, int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tat_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1, int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int count);

IppStatus ippmMul_tat_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1, int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int count);
```

Case 25: Matrix array - matrix array operation

```
IppStatus ippmMul_mama_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int count);

IppStatus ippmMul_mama_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int count);

IppStatus ippmMul_mama_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int count);
```

```
IppStatus ippmMul_mama_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mama_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp32f** ppDst, int
dstRoiShift, int dstStride1, int dstStride2, int count);

IppStatus ippmMul_mama_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp64f** ppDst, int
dstRoiShift, int dstStride1, int dstStride2, int count);
```

Case 26: Transposed matrix array - matrix array operation

```
IppStatus ippmMul_tama_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride0, int
src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp32f* pDst, int dstStride0,
int dstStride1, int dstStride2, int count);

IppStatus ippmMul_tama_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride0, int
src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp64f* pDst, int dstStride0,
int dstStride1, int dstStride2, int count);

IppStatus ippmMul_tama_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tama_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tama_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp32f** ppDst, int
dstRoiShift, int dstStride1, int dstStride2, int count);

IppStatus ippmMul_tama_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp64f** ppDst, int
dstRoiShift, int dstStride1, int dstStride2, int count);
```

Case 27: Matrix array - transposed matrix array operation

```
IppStatus ippmMul_mata_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride0, int
src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp32f* pDst, int dstStride0,
int dstStride1, int dstStride2, int count);

IppStatus ippmMul_mata_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride0, int
src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp64f* pDst, int dstStride0,
int dstStride1, int dstStride2, int count);
```

```
IppStatus ippmMul_mata_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mata_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mata_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp32f** ppDst, int
dstRoiShift, int dstStride1, int dstStride2, int count);

IppStatus ippmMul_mata_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp64f** ppDst, int
dstRoiShift, int dstStride1, int dstStride2, int count);
```

Case 28: Transposed matrix array - transposed matrix array operation

```
IppStatus ippmMul_tata_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride0, int
src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp32f* pDst, int dstStride0,
int dstStride1, int dstStride2, int count);

IppStatus ippmMul_tata_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride0, int
src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp64f* pDst, int dstStride0,
int dstStride1, int dstStride2, int count);

IppStatus ippmMul_tata_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tata_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tata_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp32f** ppDst, int
dstRoiShift, int dstStride1, int dstStride2, int count);

IppStatus ippmMul_tata_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
int src2Stride1, int src2Stride2, int src2Width, int src2Height, Ipp64f** ppDst, int
dstRoiShift, int dstStride1, int dstStride2, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).

srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pSrc1, ppSrc1	Pointer to the first source matrix or array of matrices.
src1Stride0	Stride between the matrices in the first source array.
src1Stride1	Stride between the rows in the first source matrix(ces).
src1Stride2	Stride between the elements in the first source matrix(ces).
src1RoiShift	ROI shift in the first source matrix(ces).
src1Width	Width of the first source matrix(ces).
src1Height	Height of the first source matrix(ces).
pSrc2, ppSrc2	Pointer to the second source matrix or array of matrices.
src2Stride0	Stride between the matrices in the second source array.
src2Stride1	Stride between the rows in the second source matrix(ces).
src2Stride2	Stride between the elements in the second source matrix(ces).
src2RoiShift	ROI shift in the second source matrix(ces).
src2Width	Width of the second source matrix(ces).
src2Height	Height of the second source matrix(ces).
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
val	Multiplier used in matrix scaling.
src2Len	Vector length.
width	Matrix width.
height	Matrix height.
count	Number of matrices in the array.

Description

The function `ippmMul` is declared in the `ippm.h` header file. Like all other Intel IPP matrix operating functions, this function is parameter sensitive. All input parameters that follow the function name immediately after the underscore determine the way in which the function performs and the arguments it takes, whether it is a constant, a vector, or another matrix. This implies that with every complete function name only some of the listed arguments appear in the input list, while others are omitted.

When performed on a constant and a matrix (cases 1, 3), the function scales the source matrix by multiplying each element of the source by *val,* and stores the result in *pDst*:

$$dst[i][j] = val \times src[i][j], 0 \le i < height, 0 \le j < width$$

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by the size of the data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the *`roiShift`* value is negative or not divisible by the size of the data type.
`ippStsCountMatrixErr`	Returns an error when the *`count`* value is less or equal to zero.
`ippStsSizeMatchMatrixErr`	Returns an error when the sizes of the source matrices are unsuitable.

Example

The code example below demonstrates how to use the function `ippmMul_mac_32f`.

```
IppStatus mul_mac_32f(void){
    /* Source data: 2 matrices with width=3 and height=3 */
    Ipp32f pSrc[8*3] = { 3.0f, 1.1f, 1.2f,
                         1.3f, 1.4f, 1.5f,
                         2.0f, 2.1f, 2.2f,
                         0.0f, 0.0f, 0.0f,
                         3.3f, 2.4f, 2.5f,
                         3.0f, 3.1f, 3.2f,
                         4.3f, 4.4f, 4.5f,
                         0.0f, 0.0f, 0.0f };
    /* Standard description for 2 source matrices */
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 3*sizeof(Ipp32f);
    int srcStride0 = 4*3*sizeof(Ipp32f);

    Ipp32f val=2.0;

    /* Standard description for 2 destination matrices */
    Ipp32f pDst[2*3*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);
    int dstStride0 = 9*sizeof(Ipp32f);

    int width  = 3;
    int height = 3;
    int count  = 2;

    IppStatus status = ippmMul_mac_32f((const Ipp32f*)pSrc, srcStride0,
```

```
        srcStride1, srcStride2, val, pDst, dstStride0, dstStride1,
        dstStride2, width, height, count);

    /*
    // It is recommended to check return status
    // to detect wrong input parameters, if any
    */
    if (status == ippStsNoErr) {
        printf_ma_Ipp32f("Destination matrices:", pDst, 3, 3, 2, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
    return status;
}
```

Output:

```
Destination matrices:
6.000000 2.200000 2.400000     6.600000 4.800000 5.000000
2.600000 2.800000 3.000000     6.000000 6.200000 6.400000
4.000000 4.200000 4.400000     8.600000 8.800000 9.000000
```

When performed on a constant and a transposed matrix (cases 2, 4), the function scales the transposed source matrix by multiplying each element of the source matrix by *val* and stores the result in *pDst*:

$$dst[i][j] = val \times src[j][i], 0 \le i < height, 0 \le j < width$$

Note that if the operation is performed on a transposed matrix (object type t) or a transposed matrix array (object type ta), the source matrices must have the number of columns equal to *height* and the number of rows equal to *width*.

When performed on a vector and a matrix (cases 5, 7, 9, 11), the function multiplies all elements in a row of the source matrix by the respective elements in the source vector. Done in loop through all rows in the source matrix, this operation gives the destination vector, which is stored in *pDst*. The following formula applies to all matrix rows *i* and a vector:

$$dst[i] = \sum_j src1[i][j] \times src2[j], 0 \le i < src1Height, 0 \le j < src1Width$$

Note that the number of elements in the source vector *src2Len* must be equal to *src1Width*.

The code example below demonstrates how to use the function `ippmMul_mva_32f_L`.

```
IppStatus mul_mva_32f_L(void) {
    /* Src1 matrix with width=3, height=4, Stride2=2*sizeof(Ipp32f) */
    Ipp32f pSrc1[4*6] = { 1, 0, 2, 0, 3, 0,
                          4, 0, 5, 0, 6, 0,
                          7, 0, 8, 0, 9, 0,
                          2, 0, 4, 0, 6, 0 };
    /* Standard description for Src1 */
    int src1Width   = 3;
    int src1Height  = 4;
    int src1Stride2 = 2*sizeof(Ipp32f);
    int src1Stride1 = 6*sizeof(Ipp32f);

    /* Src2 data: 2 vectors with length=3, Stride2=sizeof(Ipp32f) */
    Ipp32f src2_a[3] = { 1, 6, 3 };
    Ipp32f src2_b[3] = { 4, 5, 2 };

    /* Layout description for Src2 */
    Ipp32f* ppSrc2[2] = { src2_a, src2_b };
    int src2RoiShift = 0;
    int src2Stride2  = sizeof(Ipp32f);
    int src2Length = 3;
    /*
    // Destination vector has length=src1Height=4
    // Layout description for Dst:
    */
    Ipp32f dst[2*4];
    Ipp32f* ppDst[4] = { dst, dst+4 };
    int dstRoiShift  = 0;
    int dstStride2  = sizeof(Ipp32f);

    int count = 2;


    IppStatus status = ippmMul_mva_32f_L((const Ipp32f*)pSrc1, src1Stride1,
        src1Stride2, src1Width, src1Height, (const Ipp32f**)ppSrc2,
        src2RoiShift, src2Stride2, src2Length,
        ppDst, dstRoiShift, dstStride2, count);


    /*

    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */

    if (status == ippStsNoErr) {

        printf_va_Ipp32f("2 destination vectors:", dst, 4, 2, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }

    return status;
}
```

Output:

```
2 destination vectors:

22.000000  52.000000  82.000000  44.000000

20.000000  53.000000 86.000000  40.000000
```

When performed on a vector and a transposed matrix (cases 6, 8, 10, 12), the function multiplies all elements in a column of the source matrix by the respective elements in the source vector. Done in a loop through all columns in the source matrix, this operation gives the destination vector, which is stored in *pDst*. The following formula applies to all matrix columns *j* and a vector:

$$dst[j] = \sum_i src1[i][j] \times src2[i], 0 \le i < src1Height, 0 \le j < src1Width$$

Note that the number of elements in the source vector *src2Len* must be equal to *src1Height*.

The code example below demonstrates how to use the function `ippmMul_tva_32f`.

```
IppStatus mul_tva_32f(void) {
    /* Src1 matrix with width=3, height=4, Stride2=2*sizeof(Ipp32f) */
    Ipp32f pSrc1[4*6] = { 1, 0, 2, 0, 3, 0,
                          4, 0, 5, 0, 6, 0,
                          7, 0, 8, 0, 9, 0,
                          2, 0, 4, 0, 6, 0 };
    /* Standard description for Src1 */
    int src1Width   = 3;
    int src1Height  = 4;
    int src1Stride2 = 2*sizeof(Ipp32f);
    int src1Stride1 = 6*sizeof(Ipp32f);

    /* Src2 data: 2 vectors with length=4, Stride2=sizeof(Ipp32f) */
    Ipp32f pSrc2[2*4] = { 1, 6, 3, 2,
                          4, 5, 2, 1 };
    int src2Length  = 4;
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride0 = 4*sizeof(Ipp32f);
    /*
    // As the first operand is transposed matrix
    // destination vector has length=src1Width=3
    */
    Ipp32f pDst[2*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = 3*sizeof(Ipp32f);

    int count  = 2;

    IppStatus status = ippmMul_tva_32f((const Ipp32f*)pSrc1,
```

```
        src1Stride1, src1Stride2, src1Width, src1Height,
        (const Ipp32f*)pSrc2, src2Stride0, src2Stride2, src2Length,
        pDst, dstStride0, dstStride2, count);

    /*

    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */

    if (status == ippStsNoErr) {

        printf_va_Ipp32f("2 destination vectors:", pDst, 3, 2, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }

    return status;
}
```

Output:

```
2 destination vectors:

50.000000  64.000000  78.000000

40.000000  53.000000  66.000000
```

When performed on two matrices (cases 13, 17, 21, 25), the function multiplies the elements in a row of the first source matrix by the respective elements in a column of the second source matrix, and sums the products to obtain a new element in the destination matrix. Done in loop through all rows in the first source matrix and all columns in the second source matrix, this operation gives the whole destination matrix, which is stored in *pDst*. For an element in the destination matrix:

$$mdst[i][j] = \sum_k src1[i][k] \times src2[k][j],$$

$$0 \le i < src1Height, 0 \le j < src2Width, 0 \le k < src1Width$$

Note that the number of columns in the first source matrix *src1Width* must be equal to the number of rows in the second source matrix *src2Height*. The destination matrix has the number of columns equal to *src2Width* and the number of rows equal to *src1Height*.

ippmMul_mm4x4_32f is a special case for 4x4 matrices with *srcStride1*=4*sizeof(float) and *srcStride2*=sizeof(float).

The code example below demonstrates how to use the function ippmMul_mm_32f.

```
IppStatus mul_mm_32f(void) {
    /* Src1 matrix with width=4 and height=3 */
    Ipp32f pSrc1[3*4] = { 1, 2, 3, 4,
                          5, 6, 7, 8,
                          4, 3, 2, 1 };
    int src1Width  = 4;
```

```
    int src1Height = 3;
    int src1Stride2 = sizeof(Ipp32f);
    int src1Stride1 = 4*sizeof(Ipp32f);

    /* Src2 matrix with width=3 and height=4 */
    Ipp32f pSrc2[4*3] = { 1, 5, 4,
                          2, 6, 3,
                          3, 7, 2,
                          4, 8, 1 };
    int src2Width  = 3;
    int src2Height = 4;
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride1 = 3*sizeof(Ipp32f);

    /*
    // Destination matrix has width=src2Width=3 and height=src1Height=3
    */
    Ipp32f pDst[3*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);

    IppStatus status = ippmMul_mm_32f((const Ipp32f*)pSrc1, src1Stride1,
        src1Stride2, src1Width, src1Height, (const Ipp32f*)pSrc2,
        src2Stride1, src2Stride2, src2Width, src2Height,
        pDst, dstStride1, dstStride2);

    /*

    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */

    if (status == ippStsNoErr) {

        printf_m_Ipp32f("Destination matrix:", pDst, 3, 3, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }

    return status;
}
```

Output:

```
Destination matrix:

30.000000  70.000000  20.000000

70.000000  174.000000  0.000000

20.000000  60.000000  30.000000
```

When performed on a transposed matrix and a matrix (cases 14, 18, 22, 26), the function multiplies the elements in a column of the first source matrix by the respective elements in the column of the second source matrix, and sums the products to obtain a new element in the destination matrix. Done in loop through all columns in the first and the second source matrices, this operation gives the whole destination matrix, which is stored in *pDst*. For an element in the destination matrix:

$$dst[i][j] = \sum_k src1[k][i] \times src2[k][j],$$

$$0 \le i < src1Width, 0 \le j < src2Width, 0 \le k < src1Height$$

Note that the number of rows in the first source matrix *src1Height* must be equal to the number of rows in the second source matrix *src2Height*. The destination matrix has the number of rows equal to *src1Width* and the number of columns equal to *src2Width*.

The code example demonstrates how to use the function `ippmMul_tm_32f`. To clarify pointer descriptor for transposed matrices, see examples for ippmSub.

```
IppStatus mul_tm_32f(void) {
    /* Src1 matrix with width=2 and height=4 */
    Ipp32f pSrc1[4*2] = { 1, 2,
                          3, 4,
                          5, 6,
                          7, 8 };
    int src1Width  = 2;
    int src1Height = 4;
    int src1Stride2 = sizeof(Ipp32f);
    int src1Stride1 = 2*sizeof(Ipp32f);

    /* Src2 matrices have width=3 and height=4 */
    Ipp32f pSrc2[4*3] = { 1, 5, 4,
                          2, 6, 3,
                          3, 7, 2,
                          4, 8, 1 };
    int src2Width  = 3;
    int src2Height = 4;
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride1 = 3*sizeof(Ipp32f);
    /*
    // As the first operand is transposed matrix
    // destination matrix has width=src2Width=3 and height=src1Width=2
    */
    Ipp32f pDst[2*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);

    IppStatus status = ippmMul_tm_32f((const Ipp32f*)pSrc1, src1Stride1,
        src1Stride2, src1Width, src1Height, (const Ipp32f*)pSrc2,
        src2Stride1, src2Stride2, src2Width, src2Height,
        pDst, dstStride1, dstStride2);
    /*
    // It is recommended to check return status
    // to detect wrong input parameters, if any
    */
    if (status == ippStsNoErr) {
        printf_m_Ipp32f("Destination matrix:", pDst, 3, 2, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
    return status;
}
```

Output:

```
Destination matrix:
```

```
50.000000  114.000000  30.000000

60.000000  140.000000  40.000000
```

When performed on a matrix and a transposed matrix (cases 15, 19, 23, 27), the function multiplies the elements in a row of the first source matrix by the respective elements in the a row of the second source matrix, and sums the products to obtain a new element in the destination matrix. Done in loop through all rows in the first source matrix and all rows in the second source matrix, this operation gives the whole destination matrix, which is stored in *pDst*. For an element in the destination matrix:

$$dst[i][j] = \sum_k src1[i][k] \times src2[j][k],$$
$$0 \le i < src1Height, 0 \le j < src2Height, 0 \le k < src1Width$$

Note that the number of columns in the first source matrix *src1Width* must be equal to the number of columns in the second source matrix *src2Width*. The destination matrix has the number of rows equal to *src1Height* and the number of columns equal to *src2Height*.

When performed on two transposed matrices (cases 16, 20, 24, 28), the function multiplies the elements in a column of the first source matrix by the respective elements in a row of the second source matrix, and sums the products to obtain a new element in the destination matrix. Done in loop through all columns in the first source matrix and all rows in the second source matrix, this operation gives the whole destination matrix, which is stored in *pDst*. For an element in the destination matrix:

$$dst[i][j] = \sum_k src1[k][i] \times src2[j][k],$$
$$0 \le i < src1Width, 0 \le j < src2Height, 0 \le k < src1Height$$

Note that the number of rows in the first source matrix *src1Height* must be equal to the number of columns in the second source matrix *src2Width*. The destination matrix has the number of rows equal to *src1Width* and the number of columns equal to *src2Height*.

The code example below demonstrates how to use the function `ippmMul_tt_32f_P`.

```
IppStatus mul_tt_32f_P(void) {
    /* Src1 source data */
    Ipp32f src1[2*6] = { 9, 0, 0, 8, 0, 7,
                         0, 6, 5, 0, 0, 4 };
    /*
    // Nonzero elements of interest are referred by mask using
    // pointer descriptor: Src1 width=3, height=2
    */
    Ipp32f* ppSrc1[2*3] = { src1,   src1+3, src1+5,
                            src1+7, src1+8, src1+11 };
    int src1RoiShift = 0;
    int src1Width  = 3;
    int src1Height = 2;

    /* Src2 source data */
    Ipp32f src2[4*5] = { 0, 7, 0, 2, 0,
```

```
                         1, 0, 0, 3, 0,
                         4, 0, 5, 0, 0,
                         6, 0, 0, 0, 8 };

    /*
    // Nonzero elements of interest are referred by mask using
    // pointer descriptor: Src2 width=2, height=4
    */
    Ipp32f* ppSrc2[4*2] = { src2+1,  src2+3,
                            src2+5,  src2+8,
                            src2+10, src2+12,
                            src2+15, src2+19 };
    int src2RoiShift = 0;
    int src2Width  = 2;
    int src2Height = 4;
    /*
    // As the both operands are transposed matrices
    // destination matrix has width=src2Height=4 and height=src1Width=3
    // Pointer description for destination matrix:
    */

    Ipp32f  dst[3*4];
    Ipp32f* ppDst[3*4] = { dst,   dst+1, dst+2,  dst+3,
                           dst+4, dst+5, dst+6,  dst+7,
                           dst+8, dst+9, dst+10, dst+11 };
    int dstRoiShift = 0;

    IppStatus status = ippmMul_tt_32f_P((const Ipp32f**)ppSrc1,
        src1RoiShift, src1Width, src1Height, (const Ipp32f**)ppSrc2,
        src2RoiShift, src2Width, src2Height, ppDst, dstRoiShift);

    /*

    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */

    if (status == ippStsNoErr) {

        printf_m_Ipp32f_P("Destination matrix:", ppDst, 4, 3, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }

    return status;
}
```

Output:

```
Destination matrix:

75.000000 27.000000 66.000000 102.000000

66.000000 23.000000 57.000000 88.000000

57.000000 19.000000 48.000000 74.000000
```

Add

Adds matrix to another matrix.

Syntax

Case 1: Matrix - matrix operation

```
IppStatus ippmAdd_mm_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, const
Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride1, int
dstStride2, int width, int height);
```

```
IppStatus ippmAdd_mm_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, const
Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride1, int
dstStride2, int width, int height);
```

```
IppStatus ippmAdd_mm_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, const Ipp32f** ppSrc2,
int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int width, int height);
```

```
IppStatus ippmAdd_mm_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, const Ipp64f** ppSrc2,
int src2RoiShift, Ipp64f** pDst, int dstRoiShift, int width, int height);
```

Case 2: Transposed matrix - matrix operation

```
IppStatus ippmAdd_tm_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, const
Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride1, int
dstStride2, int width, int height);
```

```
IppStatus ippmAdd_tm_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, const
Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride1, int
dstStride2, int width, int height);
```

```
IppStatus ippmAdd_tm_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, const Ipp32f** ppSrc2,
int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int width, int height);
```

```
IppStatus ippmAdd_tm_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, const Ipp64f** ppSrc2,
int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int width, int height);
```

Case 3: Transposed matrix - transposed matrix operation

```
IppStatus ippmAdd_tt_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, const
Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride1, int
dstStride2, int width, int height);
```

```
IppStatus ippmAdd_tt_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, const
Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride1, int
dstStride2, int width, int height);
```

```
IppStatus ippmAdd_tt_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, const Ipp32f** ppSrc2,
int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int width, int height);
```

```
IppStatus ippmAdd_tt_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, const Ipp64f** ppSrc2,
int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int width, int height);
```

Case 4: Matrix array - matrix operation

```
IppStatus ippmAdd_mam_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_mam_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_mam_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);
```

```
IppStatus ippmAdd_mam_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);
```

```
IppStatus ippmAdd_mam_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_mam_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);
```

Case 5: Transposed matrix array - matrix operation

```
IppStatus ippmAdd_tam_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_tam_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_tam_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);
```

```
IppStatus ippmAdd_tam_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);
```

```
IppStatus ippmAdd_tam_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_tam_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);
```

Case 6: Matrix array - transposed matrix operation

```
IppStatus ippmAdd_mat_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f* pDst, int
dstStride0, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_mat_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f* pDst, int
dstStride0, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_mat_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int dstStride0,
int width, int height, int count);
```

```
IppStatus ippmAdd_mat_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int dstStride0,
int width, int height, int count);
```

```
IppStatus ippmAdd_mat_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1, int
src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f** ppDst, int
dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_mat_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1, int
src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f** ppDst, int
dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);
```

Case 7: Transposed matrix array - transposed matrix operation

```
IppStatus ippmAdd_tat_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f* pDst, int
dstStride0, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_tat_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f* pDst, int
dstStride0, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_tat_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int dstStride0,
int width, int height, int count);
```

```
IppStatus ippmAdd_tat_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int dstStride0,
int width, int height, int count);
```

```
IppStatus ippmAdd_tat_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1, int
src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f** ppDst, int
dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_tat_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1, int
src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f** ppDst, int
dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);
```

Case 8: Matrix array - matrix array operation

```
IppStatus ippmAdd_mama_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int
count);
```

```
IppStatus ippmAdd_mama_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int
count);
```

```
IppStatus ippmAdd_mama_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst, int dstRoiShift,
int dstStride0, int width, int height, int count);
```

```
IppStatus ippmAdd_mama_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst, int dstRoiShift,
int dstStride0, int width, int height, int count);
```

```
IppStatus ippmAdd_mama_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
int count);
```

```
IppStatus ippmAdd_mama_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
int count);
```

Case 9: Transposed matrix array - matrix array operation

```
IppStatus ippmAdd_tama_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int
count);
```

```
IppStatus ippmAdd_tama_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int
count);
```

```
IppStatus ippmAdd_tama_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst, int dstRoiShift,
int dstStride0, int width, int height, int count);
```

```
IppStatus ippmAdd_tama_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst, int dstRoiShift,
int dstStride0, int width, int height, int count);
```

```
IppStatus ippmAdd_tama_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
int count);
```

```
IppStatus ippmAdd_tama_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
int count);
```

Case 10: Transposed matrix array - transposed matrix array operation

```
IppStatus ippmAdd_tata_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int
count);
```

```
IppStatus ippmAdd_tata_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int
count);
```

```
IppStatus ippmAdd_tata_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst, int dstRoiShift,
int dstStride0, int width, int height, int count);
```

```
IppStatus ippmAdd_tata_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst, int dstRoiShift,
int dstStride0, int width, int height, int count);
```

```
IppStatus ippmAdd_tata_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
int count);
```

```
IppStatus ippmAdd_tata_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the first source matrix or array of matrices.
src1Stride0	Stride between the matrices in the first source array.
src1Stride1	Stride between the rows in the first source matrix(ces).
src1Stride2	Stride between the elements in the first source matrix(ces).
src1RoiShift	ROI shift in the first source matrix(ces).
pSrc2, ppSrc2	Pointer to the second source matrix or array of matrices.
src2Stride0	Stride between the matrices in the second source array.
src2Stride1	Stride between the rows in the second source matrix(ces).
src2Stride2	Stride between the elements in the second source matrix(ces).
src2RoiShift	ROI shift in the second source matrix(ces).
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.

dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
width	Matrix width.
height	Matrix height.
count	Number of matrices in the array.

Description

The function `ippmAdd` is declared in the `ippm.h` header file.

When performed on two matrices (cases 1, 4, 8), the function adds together the respective elements of the first and the second source matrices and stores the result in *pDst*:

dst[i][j] = src1[i][j] + src2[i][j],

$0 \le i < height$, $0 \le j < width$.

When performed on a transposed matrix and a matrix (cases 2, 5, 9), the function adds together the respective elements of the transposed matrix and the second source matrix and stores the result in *pDst*:

dst[i][j] = src1[j][i] + src2[i][j],

$0 \le i < height$, $0 \le j < width$.

Note that the first source matrix must have the number of rows equal to *width* and the number of columns equal to *height*.

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by the size of the data type.
ippStsRoiShiftMatrixErr	Returns an error when the *roiShift* value is negative or not divisible by the size of the data type.
ippStsCountMatrixErr	Returns an error when the *count* value is less or equal to zero.

Example

The code example below demonstrates how to use the function `ippmAdd_tm_32f`. To clarify pointer descriptor for transposed matrices, see examples for `ippmSub`.

```
IppStatus add_tm_32f(void) {
    /* Src1 matrix with width=4 and height=3 */
    Ipp32f pSrc1[3*4] = { 1, 2, 3, 4,
                          5, 6, 7, 8,
                          4, 3, 2, 1 };
    int src1Stride2 = sizeof(Ipp32f);
    int src1Stride1 = 4*sizeof(Ipp32f);

    /* Src2 and Dst matrices have width=3 and height=4 */
    Ipp32f pSrc2[4*3] = { 1, 5, 4,
```

```
                        2, 6, 3,
                        3, 7, 2,
                        4, 8, 1 };
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride1 = 3*sizeof(Ipp32f);

    Ipp32f pDst[4*3]; /* Destination location */
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);


    int width  = 3;
    int height = 4;

    IppStatus status = ippmAdd_tm_32f((const Ipp32f*)pSrc1, src1Stride1,
         src1Stride2, (const Ipp32f*)pSrc2, src2Stride1, src2Stride2,
         pDst, dstStride1, dstStride2, width, height);

    /*

    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */

    if (status == ippStsNoErr) {

        printf_m_Ipp32f("Destination matrix:", pDst, 3, 4, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }

    return status;
}
```

Output:

```
Destination matrix:

2.000000  10.000000  8.000000

4.000000  12.000000  6.000000

6.000000  14.000000  4.000000

8.000000  16.000000  2.000000
```

When performed on a matrix array and a transposed matrix (case 6), the function adds together the respective elements of the first matrix in the array and the transposed matrix and stores the result in *pDst*:

dst[i][j] = src1[i][j] + src2[j][i],

$0 \le i < height$, $0 \le j < width$.

Note that the second source matrix must have the number of rows equal to *width* and the number of columns equal to *height*.

When performed on two transposed matrices (cases 3, 7, 10), the function adds together the respective elements of the first and the second transposed matrices and stores the result in *pDst*:

dst[i][j] = src1[j][i] + src2[j][i],

0 ≤ *i* < *height*, 0 ≤ *j* < *width*.

Note that both source matrices must have the number of rows equal to *width* and the number of columns equal to *height*.

The code example below demonstrates how to use the function `ippmAdd_tt_32f`. To clarify pointer descriptor for transposed matrices, see examples for `ippmSub`.

```
IppStatus add_tt_32f(void) {
    /* Src1 and Src2 matrices have width=4 and height=3 */
    Ipp32f pSrc1[3*4] = { 1, 2, 3, 1,
                          4, 5, 6, 1,
                          7, 8, 9, 1 };
    int src1Stride2 = sizeof(Ipp32f);
    int src1Stride1 = 4*sizeof(Ipp32f);

    Ipp32f pSrc2[3*4] = { 9, 8, 7, -1,
                          6, 5, 4, -1,
                          3, 2, 1, -1 };
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride1 = 4*sizeof(Ipp32f);

    /* Dst matrices have width=3 and height=4 */
    Ipp32f pDst[4*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);


    int width  = 3;
    int height = 4;
    IppStatus status = ippmAdd_tt_32f((const Ipp32f*)pSrc1, src1Stride1,
         src1Stride2, (const Ipp32f*)pSrc2, src2Stride1, src2Stride2,
         pDst, dstStride1, dstStride2, width, height);


    /*

    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */

    if(status == ippStsNoErr){

        printf_m_Ipp32f("Destination matrix:", pDst, 3, 4, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }

    return status;
}
```

Output:

```
Destination matrix:

10.000000  10.000000  10.000000
```

```
10.000000  10.000000  10.000000
10.000000  10.000000  10.000000
0.000000   0.000000   0.000000
```

Sub

Subtracts matrix from another matrix.

Syntax

Case 1: Matrix - matrix operation

```
IppStatus ippmSub_mm_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, const
Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride1, int
dstStride2, int width, int height);
```

```
IppStatus ippmSub_mm_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, const
Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride1, int
dstStride2, int width, int height);
```

```
IppStatus ippmSub_mm_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, const Ipp32f** ppSrc2,
int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int width, int height);
```

```
IppStatus ippmSub_mm_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, const Ipp64f** ppSrc2,
int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int width, int height);
```

Case 2: Transposed matrix - matrix operation

```
IppStatus ippmSub_tm_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, const
Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride1, int
dstStride2, int width, int height);
```

```
IppStatus ippmSub_tm_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, const
Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride1, int
dstStride2, int width, int height);
```

```
IppStatus ippmSub_tm_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, const Ipp32f** ppSrc2,
int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int width, int height);
```

```
IppStatus ippmSub_tm_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, const Ipp64f** ppSrc2,
int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int width, int height);
```

Case 3: Matrix - transposed matrix operation

```
IppStatus ippmSub_mt_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, const
Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride1, int
dstStride2, int width, int height);
```

```
IppStatus ippmSub_mt_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2, const
Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride1, int
dstStride2, int width, int height);
```

```
IppStatus ippmSub_mt_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, const Ipp32f** ppSrc2,
int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int width, int height);
```

IppStatus ippmSub_mt_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *width*, int *height*);

Case 4: Transposed matrix - transposed matrix operation

IppStatus ippmSub_tt_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride1*, int *dstStride2*, int *width*, int *height*);

IppStatus ippmSub_tt_64f(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride1*, int *dstStride2*, int *width*, int *height*);

IppStatus ippmSub_tt_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *width*, int *height*);

IppStatus ippmSub_tt_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *width*, int *height*);

Case 5: Matrix - matrix array operation

IppStatus ippmSub_mma_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride0*, int *src2Stride1*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mma_64f(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride0*, int *src2Stride1*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mma_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mma_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mma_32f_L(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride1*, int *src2Stride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mma_64f_L(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride1*, int *src2Stride2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

Case 6: Transposed matrix - matrix array operation

IppStatus ippmSub_tma_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride0*, int *src2Stride1*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tma_64f(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride0*, int *src2Stride1*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tma_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tma_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tma_32f_L(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride1*, int *src2Stride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tma_64f_L(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride1*, int *src2Stride2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

Case 7: Matrix - transposed matrix array operation

IppStatus ippmSub_mta_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride0*, int *src2Stride1*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mta_64f(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride0*, int *src2Stride1*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mta_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mta_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mta_32f_L(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride1*, int *src2Stride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mta_64f_L(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride1*, int *src2Stride2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

Case 8: Transposed matrix - transposed matrix array operation

IppStatus ippmSub_tta_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride0*, int *src2Stride1*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tta_64f(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride0*, int *src2Stride1*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tta_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tta_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tta_32f_L(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride1*, int *src2Stride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tta_64f_L(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride1*, int *src2Stride2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

Case 9: Matrix array - matrix operation

IppStatus ippmSub_mam_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mam_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mam_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mam_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mam_32f_L(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mam_64f_L(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

Case 10: Transposed matrix array - matrix operation

IIppStatus ippmSub_tam_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tam_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tam_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tam_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tam_32f_L(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tam_64f_L(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

Case 11: Matrix array - transposed matrix operation

IppStatus ippmSub_mat_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mat_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mat_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mat_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mat_32f_L(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_mat_64f_L(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

Case 12: Transposed matrix array - transposed matrix operation

IppStatus ippmSub_tat_32f(const Ipp32f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tat_64f(const Ipp64f* *pSrc1*, int *src1Stride0*, int *src1Stride1*, int *src1Stride2*, const Ipp64f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tat_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tat_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Stride0*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *width*, int *height*, int *count*);

IppStatus ippmSub_tat_32f_L(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Stride1*, int *src1Stride2*, const Ipp32f* *pSrc2*, int *src2Stride1*, int *src2Stride2*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, int *width*, int *height*, int *count*);

```
IppStatus ippmSub_tat_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);
```

Case 13: Matrix array - matrix array operation

```
IppStatus ippmSub_mama_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_mama_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_mama_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_mama_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_mama_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1, int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_mama_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1, int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height, int count);
```

Case 14: Transposed matrix array - matrix array operation

```
IppStatus ippmSub_tama_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_tama_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_tama_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_tama_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);
```

```
IppStatus ippmSub_tama_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
int count);
```

```
IppStatus ippmSub_tama_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
int count);
```

Case 15: Matrix array - transposed matrix array operation

```
IppStatus ippmSub_mata_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int
count);
```

```
IppStatus ippmSub_mata_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int
count);
```

```
IppStatus ippmSub_mata_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst, int dstRoiShift,
int dstStride0, int width, int height, int count);
```

```
IppStatus ippmSub_mata_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst, int dstRoiShift,
int dstStride0, int width, int height, int count);
```

```
IppStatus ippmSub_mata_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
int count);
```

```
IppStatus ippmSub_mata_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
int count);
```

Case 16: Transposed matrix array - transposed matrix array operation

```
IppStatus ippmSub_tata_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int
count);
```

```
IppStatus ippmSub_tata_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1, int
src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int
count);
```

```
IppStatus ippmSub_tata_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst, int dstRoiShift,
int dstStride0, int width, int height, int count);
```

```
IppStatus ippmSub_tata_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride0,
const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst, int dstRoiShift,
int dstStride0, int width, int height, int count);

IppStatus ippmSub_tata_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
int count);

IppStatus ippmSub_tata_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int src1Stride1,
int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the first source matrix or array of matrices.
src1Stride0	Stride between the matrices in the first source array.
src1Stride1	Stride between the rows in the first source matrix(ces).
src1Stride2	Stride between the elements in the first source matrix(ces).
src1RoiShift	ROI shift in the first source matrix(ces).
pSrc2, ppSrc2	Pointer to the second source matrix or array of matrices.
src2Stride0	Stride between the matrices in the second source array.
src2Stride1	Stride between the rows in the second source matrix(ces).
src2Stride2	Stride between the elements in the second source matrix(ces).
src2RoiShift	ROI shift in the second source matrix(ces).
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
width	Matrix width.
height	Matrix height.
count	Number of matrices in the array.

Description

The function `ippmSub` is declared in the `ippm.h` header file.

When performed on two matrices (cases 1, 5, 9, 13), the function subtracts an element of the second source matrix from the respective element of the first source matrix and stores the result in *pDst*. This operation is done in loop through all matrix elements:

dst[i][j] = src1[i][j] - src2[i][j],

$0 \le i < height$, $0 \le j < width$.

When performed on a transposed matrix and a matrix (cases 2, 6, 10, 14), the function subtracts an element of the second source matrix from the respective element of the first source matrix and stores the result in *pDst*. This operation is done in loop through all matrix elements:

dst[i][j] = src1[j][i] - src2[i][j],

$0 \le i < height$, $0 \le j < width$.

Note that the transposed matrix must have the number of rows equal to *width* and the number of columns equal to *height*.

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by the size of the data type.
ippStsRoiShiftMatrixErr	Returns an error when the *roiShift* value is negative or not divisible by the size of the data type.
ippStsCountMatrixErr	Returns an error when the *count* value is less or equal to zero.

Example

The code example below demonstrates how to use the function ippmSub_tm_32f_P. To clarify other cases, see examples for the ippmAdd function.

```
IppStatus sub_tm_32f_P(void) {
    /* Src1 source data */
    Ipp32f src1[2*6] = { 9, 0, 0, 8, 0, 7,
                         0, 6, 5, 0, 0, 4 };
    /*
    // Nonzero elements of interest are referred by mask using
    // pointer descriptor: Src1 width=3, height=2
    */
    Ipp32f* ppSrc1[2*3] = { src1,   src1+3, src1+5,
                            src1+7, src1+8, src1+11 };

    int src1RoiShift = 0;

    /* Src2 source data */
    Ipp32f src2[3*6] = { 7, 0, 0, 0, 0, 1,
                         0, 2, 1, 0, 0, 0,
                         0, 0, 4, 0, 3, 0};

    /*
    // Nonzero elements of interest are referred by mask using
    // Pointer descriptor: Src2 width=2, height=3
    */
    Ipp32f* ppSrc2[3*2] = { src2,    src2+5,
                            src2+7,  src2+8,
                            src2+14, src2+16 };
    int src2RoiShift = 0;

    /*
    // Pointer description for destination matrix:
    // Dst width=2, height=3
```

```
    */
    Ipp32f  dst[3*2];
    Ipp32f* ppDst[3*2] = { dst,   dst+1,
                           dst+2, dst+3,
                           dst+4, dst+5 };

    int dstRoiShift = 0;
    int width  = 2;
    int height = 3;


    IppStatus status = ippmSub_tm_32f_P((const Ipp32f**)ppSrc1,
        src1RoiShift, (const Ipp32f**)ppSrc2, src2RoiShift,
        ppDst, dstRoiShift, width, height );

    /*

    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */

    if(status == ippStsNoErr){

        printf_m_Ipp32f_P("Destination matrix:", ppDst, 2, 3, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }

    return status;
}
```

Output:

```
Destination matrix:

2.000000  5.000000

6.000000  4.000000

3.000000  1.000000
```

When performed on a matrix array and a transposed matrix (cases 3, 7, 11, 15), the function subtracts an element of the transposed source matrix from the respective element of the first source matrix and stores the result in *pDst*. This operation is done in loop through all matrix elements:

dst[i][j] = src1[i][j] - src2[j][i],

$0 \le i < height$, $0 \le j < width$.

Note that the transposed matrix must have the number of rows equal to *width* and the number of columns equal to *height*.

When performed on two transposed matrices (cases 4, 8, 12, 16), the function subtracts an element of the second transposed matrix from the respective element of the first transposed matrix and stores the result in *pDst*. This operation is done in loop through all matrix elements:

dst[i][j] = src1[j][i] - src2[j][i],

$0 \leq i < height$, $0 \leq j < width$.

Note that both transposed matrices must have the number of rows equal to *width* and the number of columns equal to *height*.

The code example below demonstrates how to use the function `ippmSub_tt_32f_P`.

```
IppStatus sub_tt_32f_P(void) {
    /* Src1 source data */
    Ipp32f src1[2*6] = { 9, 0, 0, 8, 0, 7,
                         0, 6, 5, 0, 0, 4 };
    /*
    // Nonzero elements of interest are referred by mask using
    // pointer descriptor: Src1 width=3, height=2
    */
    Ipp32f* ppSrc1[2*3] = { src1,   src1+3, src1+5,
                            src1+7, src1+8, src1+11 };

    int src1RoiShift = 0;

    /* Src2 source data */
    Ipp32f src2[2*6] = { 0, 7, 0, 2, 4, 0,
                         1, 1, 0, 3, 0, 0 };


    /*
    // Nonzero elements of interest are referred by mask using
    // pointer descriptor: Src2 width=3, height=2
    */
    Ipp32f* ppSrc2[2*3] = { src2+1, src2+3, src2+4,
                            src2+6, src2+7, src2+9 };
    int src2RoiShift = 0;
    /*
    // Pointer description for destination matrix:
    // Dst width=2, height=3
    */
    Ipp32f  dst[3*2];
    Ipp32f* ppDst[3*2] = { dst,   dst+1,
                           dst+2, dst+3,
                           dst+4, dst+5 };
    int dstRoiShift = 0;
```

```
    int width  = 2;
    int height = 3;

    IppStatus status = ippmSub_tt_32f_P((const Ipp32f**)ppSrc1,
        src1RoiShift, (const Ipp32f**)ppSrc2, src2RoiShift,
        ppDst, dstRoiShift, width, height );

    /*

    // It is recommended to check return status

    // to detect wrong input parameters, if any

    */

    if(status == ippStsNoErr){

        printf_m_Ipp32f_P("Destination matrix:", ppDst, 2, 3, status);

    } else {

        printf("Function returns status: %s \n", ippGetStatusString(status));

    }

    return status;
}
```

Output:

```
Destination matrix:

2.000000  5.000000

6.000000  4.000000

3.000000  1.000000
```

Gaxpy

Performs the "gaxpy" operation on a matrix.

Syntax

Case 1: Matrix - vector - vector operation

IppStatus ippmGaxpy_mv_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride2*, int *src2Len*, const Ipp32f* *pSrc3*, int *src2Stride3*, int *src3Len*, Ipp32f* *pDst*, int *dstStride2*);

IppStatus ippmGaxpy_mv_64f(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride2*, int *src2Len*, const Ipp64f* *pSrc3*, int *src3Stride2*, int *src3Len*, Ipp64f* *pDst*, int *dstStride2*);

IppStatus ippmGaxpy_mv_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Len*, const Ipp32f** *ppSrc3*, int *src3RoiShift*, int *src3Len*, Ipp32f** *ppDst*, int *dstRoiShift*);

IppStatus ippmGaxpy_mv_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Len*, const Ipp64f** *ppSrc3*, int *src3RoiShift*, int *src3Len*, Ipp64f** *ppDst*, int *dstRoiShift*);

Case 2: Matrix - vector array - vector array operation

IppStatus ippmGaxpy_mva_32f(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride0*, int *src2Stride2*, int *src2Len*, const Ipp32f* *pSrc3*, int *src3Stride0*, int *src3Stride2*, int *src3Len*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *count*);

IppStatus ippmGaxpy_mva_64f(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride0*, int *src2Stride2*, int *src2Len*, const Ipp64f* *pSrc3*, int *src3Stride0*, int *src3Stride2*, int *src3Len*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *count*);

IppStatus ippmGaxpy_mva_32f_P(const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, int *src2Len*, const Ipp32f** *ppSrc3*, int *src3RoiShift*, int *src3Stride0*, int *src3Len*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmGaxpy_mva_64f_P(const Ipp64f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, int *src2Len*, const Ipp64f** *ppSrc3*, int *src3RoiShift*, int *src3Stride0*, int *src3Len*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

IppStatus ippmGaxpy_mva_32f_L(const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride2*, int *src2Len*, const Ipp32f** *ppSrc3*, int *src3RoiShift*, int *src3Stride2*, int *src3Len*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *count*);

IppStatus ippmGaxpy_mva_64f_L(const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f** *ppSrc2*, int *src2RoiShift*, int *src2Stride2*, int *src2Len*, const Ipp64f** *ppSrc3*, int *src3RoiShift*, int *src3Stride2*, int *src3Len*, Ipp64f** *ppDst*, int *dstRoiShift*, int *dstStride2*, int *count*;)

Case 3: Matrix - vector array - vector operation

IppStatus ippmGaxpy_mvav_32f (const Ipp32f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp32f* *pSrc2*, int *src2Stride0*, int *src2Stride2*, int *src2Len*, const Ipp32f* *pSrc3*, int *src3Stride2*, int *src3Len*, Ipp32f* *pDst*, int *dstStride0*, int *dstStride2*, int *count*);

IppStatus ippmGaxpy_mvav_64f (const Ipp64f* *pSrc1*, int *src1Stride1*, int *src1Stride2*, int *src1Width*, int *src1Height*, const Ipp64f* *pSrc2*, int *src2Stride0*, int *src2Stride2*, int *src2Len*, const Ipp64f* *pSrc3*, int *src3Stride2*, int *src3Len*, Ipp64f* *pDst*, int *dstStride0*, int *dstStride2*, int *count*);

IppStatus ippmGaxpy_mvav_32f_P (const Ipp32f** *ppSrc1*, int *src1RoiShift*, int *src1Width*, int *src1Height*, const Ipp32f** *ppSrc2*, int *src2RoiShift*, int *src2Stride0*, int *src2Len*, const Ipp32f** *ppSrc3*, int *src3RoiShift*, int *src3Len*, Ipp32f** *ppDst*, int *dstRoiShift*, int *dstStride0*, int *count*);

```
IppStatus ippmGaxpy_mvav_64f_P (const Ipp64f** ppSrc1, int src1RoiShift, int src1Width,
int src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, int src2Len,
const Ipp64f** ppSrc3, int src3RoiShift, int src3Len, Ipp64f** ppDst, int dstRoiShift,
int dstStride0, int count);
```

```
IppStatus ippmGaxpy_mvav_32f_L (const Ipp32f* pSrc1, int src1Stride1, int src1Stride2,
int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2,
int src2Len, const Ipp32f* pSrc3, int src3Stride2, int src3Len, Ipp32f** ppDst, int
dstRoiShift, int dstStride2, int count);
```

```
IppStatus ippmGaxpy_mvav_64f_L (const Ipp64f* pSrc1, int src1Stride1, int src1Stride2,
int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2,
int src2Len, const Ipp64f* pSrc3, int src3Stride2, int src3Len, Ipp64f** ppDst, int
dstRoiShift, int dstStride2, int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the source matrix or array of matrices.
src1Stride0	Stride between the matrices in the source matrix array.
src1Stride1	Stride between the rows in the source matrix(ces).
src1Stride2	Stride between the elements in the source matrix(ces).
src1RoiShift	ROI shift in the source matrix(ces).
src1Width	Matrix width.
src1Height	Matrix height.
pSrc2, ppSrc2	Pointer to the first source vector or array of vectors.
src2Stride0	Stride between the vectors in the first source vector array.
src2Stride2	Stride between the elements in the first source vector(s).
src2RoiShift	ROI shift in the first source vector(s).
src2Len	Length of the first source vector(s).
pSrc3, ppSrc3	Pointer to the second source vector or array of vectors.
src3Stride0	Stride between the vectors in the second source vector array.
src3Stride2	Stride between the elements in the second source vector(s).
src3RoiShift	ROI shift in the second source vector(s).
src3Len	Length of the second source vector(s).
pDst, ppDst	Pointer to the destination vector or array of vectors.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector(s).
dstRoiShift	ROI shift in the destination vector(s).
count	Number of objects in the array.

Description

The function `ippmGaxpy` is declared in the `ippm.h` header file. The function performs the “gaxpy” operation on a matrix by

- composing dot product of a row in the first source matrix and the second source vector
- adding the result to the respective element in the third source vector
- storing the result in the respective element in the destination:

$$dst[i] = \sum_{i} (src1[i][j] \cdot src2[j]) + src3[i], 0 \le i < src1Height, 0 \le j < src1Width$$

The function iterates consequently through all the rows of the matrices in question.

To clarify how the function operates on arrays of vectors, see the “Operations with arrays of objects” section in the Getting Started chapter.

Note that the number of elements in the second source vector *src2Len* must be equal to *src1Width* and the number of elements in the third source vector *src3Len* must be equal to *src1Height*.

For the Gaxpy function, having three source operands *src1, src2, and src3*, *by default,* the type of the object *src3* is not specified in the function name if *src2* and *src3* objects are of the same type. The function name contains specifications of all the three object types if *src2* and *src3* have different types.

For example,

ippmGaxpy_mv, default, both *src2* and *src3* operands are vectors (type v).

ippmGaxpy_mva, default, both *src2* and *src3* operands are vector arrays (type va).

ippmGaxpy_mvav, the operand *src*2 is a vector array (type va) and *src3* is a vector (type v).

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible be the size of the data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not divisible be the size of the data type.
ippStsCountMatrixErr	Returns an error when the *count* value is less or equal to zero.
ippStsSizeMatchMatrixErr	Returns an error when the sizes of the source matrices are unsuitable.

Example

The code example below demonstrates how to use the function ippmGaxpy_mva_32f.

```
IppStatus gaxpy_mva_32f(void) {
    /* Src1 matrix with width=3, height=4, Stride2=2*sizeof(Ipp32f) */
    Ipp32f pSrc1[4*6] = { 1, 0, 2, 0, 3, 0,
                          4, 0, 5, 0, 6, 0,
                          7, 0, 8, 0, 9, 0,
                          2, 0, 4, 0, 6, 0 };
    /* Src2 data: 2 vectors with length=3, Stride2=sizeof(Ipp32f) */
    Ipp32f pSrc2[2*3] = { 1, 6, 3,
                          4, 5, 2 };
```

```
    /* Src3 data: 2 vectors with length=4, Stride2=sizeof(Ipp32f) */
    Ipp32f pSrc3[2*4] = { -11, -16, -13, -17,
                           -4, -15, -12, -18 };
    /* Standard description for Src1 */
    int src1Width   = 3;
    int src1Height  = 4;
    int src1Stride2 = 2 * sizeof(Ipp32f);
    int src1Stride1 = 6 * sizeof(Ipp32f);

    /* Standard description for Src2 */
    int src2Length = 3;
    int src2Stride2  = sizeof(Ipp32f);
    int src2Stride0  = 3*sizeof(Ipp32f);


    /* Standard description for Src3 */
    int src3Length = 4;
    int src3Stride2  = sizeof(Ipp32f);
    int src3Stride0  = 4*sizeof(Ipp32f);

    /*
    // Destination vector has length=src1Height=src3Length=4
    // Standard description for Dst:
    */
    Ipp32f pDst[2*4];
    int dstStride2  = sizeof(Ipp32f);
    int dstStride0  = 4*sizeof(Ipp32f);

    int count  = 2;



    IppStatus status = ippmGaxpy_mva_32f((const Ipp32f*)pSrc1,
        src1Stride1, src1Stride2, src1Width, src1Height,
        (const Ipp32f*)pSrc2, src2Stride0, src2Stride2, src2Length,
        (const Ipp32f*)pSrc3, src3Stride0, src3Stride2, src3Length,
        pDst, dstStride0, dstStride2, count);

    /*

    // It is recommended to check return status
    // to detect wrong input parameters, if any
    */

    if(status == ippStsNoErr){

        printf_va_Ipp32f("2 destination vectors:", pDst, 4, 2, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
    return status;
}
```

Output:

```
2 destination vectors:

11.000000  36.000000  69.000000  27.000000

16.000000  38.000000  74.000000  22.000000
```

The code example below demonstrates how to use the function `ippmGaxpy_mvav_L_32f`.

```
IppStatus gaxpy_mvav_32f_L(void) {
    /* Src1 data: matrix with width=3, height=4, Stride2=2*sizeof(Ipp32f) */
    Ipp32f pSrc1[4*6] = { 1, 0, 2, 0, 3, 0,
                          4, 0, 5, 0, 6, 0,
                          7, 0, 8, 0, 9, 0,
                          2, 0, 4, 0, 6, 0 };

    /*
    // Src2 data: 2 vectors with length=3, Stride2=sizeof(Ipp32f)
    // Vectors are spaced in the memory irregularly
    */
    Ipp32f src2_a[3] = { 1, 6, 3 };
    Ipp32f src2_b[3] = { 4, 5, 2 };

    /* Src3 data: 1 vectors with length=4, Stride2=sizeof(Ipp32f) */
    Ipp32f pSrc3[4] = { -11, -16, -13, -17 };

    /* Standard description for Src1 */
    int src1Width   = 3;
    int src1Height  = 4;
    int src1Stride2 = 2 * sizeof(Ipp32f);
    int src1Stride1 = 6 * sizeof(Ipp32f);

    /* Layout description for Src2 */
    Ipp32f* ppSrc2[2] = { src2_a, src2_b };
    int src2Length = 3;
    int src2Stride2  = sizeof(Ipp32f);
    int src2RoiShift = 0;

    /* Standard description for Src3 */
    int src3Length = 4;
    int src3Stride2  = sizeof(Ipp32f);


    /*
    // Layout description for Dst:
    // Destination vector has length=src1Height=src3Length=4
    */
    Ipp32f dst[2*4];
    Ipp32f* ppDst[2] = { dst, dst+4};

    int dstStride2  = sizeof(Ipp32f);
    int dstRoiShift = 0;

    int count  = 2;

    IppStatus status = ippmGaxpy_mvav_32f_L ((const Ipp32f*)pSrc1,
        src1Stride1, src1Stride2, src1Width, src1Height,
        (const Ipp32f**)ppSrc2, src2RoiShift, src2Stride2, src2Length,
        (const Ipp32f*)pSrc3, src3Stride2, src3Length,
        ppDst, dstRoiShift, dstStride2, count);

    /*
    // It is kindly recommended to check return status
    // to avoid wrong input parameters
    */
    if(status == ippStsOk){
        printf_va_Ipp32f("2 destination vectors:", dst, 4, 2, status);
    } else {
        printf("Function returns status: %s \n",
            ippGetStatusString(status));
```

```
    }
    return status;
}
```

Output:

```
2 destination vectors:
11.000000  36.000000  69.000000  27.000000
16.000000  38.000000  74.000000  22.000000
```

AffineTransform3DH

Performs an arbitrary affine transformation with an array of 3D vectors in the Homogeneous coordinate space.

Syntax

```
IppStatus ippmAffineTransform3DH_mva_32f (const Ipp32f* pSrc1, int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride2, int count);
```

Parameters

pSrc1	Pointer to the source matrix of size 4x4.
src1Stride1	Stride between the rows in the source matrix.
src1Stride2	Stride between the elements in the source matrix.
pSrc2	Pointer to the source array of 3D vectors.
src2Stride0	Stride between the vectors in the source 3D vector array.
src2Stride2	Stride between the elements in the source 3D vector.
pDst	Pointer to the destination array of 3D vectors.
dstStride0	Stride between the 3D vectors in the destination array.
dstStride2	Stride between the elements in the destination 3D vector.
count	The number of 3D vectors in the array.

Description

The function `ippmAffineTransform3DH` is declared in the `ippm.h` header file. The function performs an arbitrary affine transformation (possibly, a degenerate projective transformation) with an array of 3D vectors in the Homogeneous coordinate space. The transformation is represented with the 4x4 matrix pointed by *pSrc1*. Input and output 3D vector arrays are located at the *pSrc2* and *pDst* addresses, respectively. The number of 3D vectors in the array equals *count*.

During this transformation, three-dimensional Cartesian coordinates are first transformed into a homogeneous coordinate space, in which the affine transformation is then applied to the vectors using matrix-vector multiplication. After that, the inverse transformation from the Homogeneous to the Cartesian coordinate system is performed.

In more detail, the function performs the following matrix-vector operation to each 3D vector in the source vector array:

$$\begin{bmatrix} X' \\ Y' \\ Z' \\ W \end{bmatrix} = \begin{bmatrix} t_{11} & t_{12} & t_{13} & t_{14} \\ t_{21} & t_{22} & t_{23} & t_{24} \\ t_{31} & t_{32} & t_{33} & t_{34} \\ t_{41} & t_{42} & t_{43} & t_{44} \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

that is

$X' = t_{11}*X + t_{12}*Y + t_{13}*Z + t_{14}$

$Y' = t_{21}*X + t_{22}*Y + t_{23}*Z + t_{24}$

$Z' = t_{31}*X + t_{32}*Y + t_{33}*Z + t_{34}$

$W = t_{41}*X + t_{42}*Y + t_{43}*Z + t_{44}$,

and then the following vector transformation:

X' = X'/W

Y' = Y'/W

Z' = Z'/W

where

t_{ij}, $0 \le i < 4$, $0 \le j < 4$, are elements of the source 4x4 matrix *Src1*,

(X, Y, Z) is the source 3D vector *Src2,*

(X',Y',Z') is the destination 3D vector *Dst.*

If W is equal to zero, all the 3D coordinates of the destineation vector are set to IPP_MAXABS, that is, the maximum machine number for the given data type (defined in the ippdefs.h header file), and the function returns the ippStsDivByZeroErr status. However the calculations continue for the remaining 3D vectors in the source array.

Return Values

ippStsOk	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one input pointer is NULL.
ippStsStrideMatrixErr	Indicates an error when a stride value is not positive or not divisible by the size of the data type.
ippStsCountMatrixErr	Indicates an error when the *count* value is less or equal to 0.
ippStsDivByZeroErr	Indicates an error when W is equal to 0 for at least one 3D vector.

Example

The code example below demonstrates how to use the function `ippmAffineTransform3DH_mva_32f`.

```
IppStatus  AffineTransform3DH_mva_32f(void) {
    /* Transformation matrix Src1 with width=4, height=4 as default */
    Ipp32f pSrc1[4*4] = { 1, 2, 3, 2,
                          4, 5, 6, 5,
                          7, 8, 9, 6,
                          2, 4, 6, 1 };

    /* Standard description for Src1 */
    int src1Stride1 = 4 * sizeof(Ipp32f);
    int src1Stride2 = sizeof(Ipp32f);

    /* Src2 data: two 3D vectors with length=3 as default */
    Ipp32f pSrc2[2*3] = { 1, 6, 3,
                          4, 5, 2 };

    /* Standard description for Src2 */
    int src2Stride0  = 3*sizeof(Ipp32f);
    int src2Stride2  = sizeof(Ipp32f);


    /*
    // Destination is two 3D vectors
    // Standard description for Dst:
    */
    Ipp32f pDst[2*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = 3*sizeof(Ipp32f);

    int count = 2;
    IppStatus status = ippmAffineTransform3DH_mva_32f(
```

```
        (const Ipp32f*)pSrc1, src1Stride1, src1Stride2,
        (const Ipp32f*)pSrc2, src2Stride0, src2Stride2,
        pDst, dstStride0, dstStride2, count);

    /*
    // It is required for AffineTransform3DH function to check return status
    // for catching wrong result in case of invalid input data

    */
    if(status == ippStsOk){
        printf_va_Ipp32f("Two destination 3D vectors:", pDst, 3, 2, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
    return status;
}
```

Output:

```
Two destination 3D vectors:
0.533333  1.266667  1.955556
0.536585  1.414634  2.243902
```

Linear System Solution Functions 6

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) functions for small matrices that decompose square matrices and solve system of linear equations by back substitution.

Linear System Solution functions

Function Base Name	Operation
`LUDecomp`	Decomposes square matrix into product of upper and lower triangular matrices.
`LUBackSubst`	Solves system of linear equations with LU-factored square matrix.
`CholeskyDecomp`	Performs Cholesky decomposition of a symmetric positive definite square matrix.
`CholeskyBackSubst`	Solves system of linear equations using the Cholesky triangular factor.

LUDecomp

Decomposes square matrix into product of upper and lower triangular matrices.

Syntax

Case 1: Matrix operation

```
IppStatus ippmLUDecomp_m_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2, int* pDstIndex, Ipp32f* pDst, int dstStride1, int dstStride2, int widthHeight);
```

```
IppStatus ippmLUDecomp_m_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2, int* pDstIndex, Ipp64f* pDst, int dstStride1, int dstStride2, int widthHeight);
```

```
IppStatus ippmLUDecomp_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int* pDstIndex, Ipp32f** ppDst, int dstRoiShift, int widthHeight);
```

```
IppStatus ippmLUDecomp_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int* pDstIndex, Ipp64f** ppDst, int dstRoiShift, int widthHeight);
```

Case 2: Matrix array operation

```
IppStatus ippmLUDecomp_ma_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1, int srcStride2, int* pDstIndex, Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmLUDecomp_ma_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1, int srcStride2, int* pDstIndex, Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmLUDecomp_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0,
int* pDstIndex, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int widthHeight, int
count);
```

```
IppStatus ippmLUDecomp_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0,
int* pDstIndex, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int widthHeight, int
count);
```

```
IppStatus ippmLUDecomp_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1,
int srcStride2, int* pDstIndex, Ipp32f** ppDst, int dstRoiShift, int dstStride1, int
dstStride2, int widthHeight, int count);
```

```
IppStatus ippmLUDecomp_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1,
int srcStride2, int* pDstIndex, Ipp64f** ppDst, int dstRoiShift, int dstStride1, int
dstStride2, int widthHeight, int count);
```

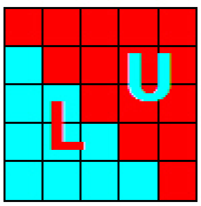
Parameters

pSrc, *ppSrc*	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDstIndex	Pointer to array of pivot indices, where row *i* interchanges with row *index(i)*. The array size can be more than or equal to *widthHeight*. If the operation is performed on an array of matrices, the size of the array of indices must be more than or equal to *count* **widthHeight*.
pDst, *ppDst*	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
widthHeight	Size of the square matrix.
count	Number of matrices in the array.

Description

The function `ippmLUDecomp` is declared in the `ippm.h` header file. The function represents the source matrix *pSrc* or *ppSrc* as a product of two matrices *L* and *U*, where *L* is the lower triangular with unit diagonal elements and *U* is the upper triangular (see Figure "LU Decomposition Matrix Storage"). Both *L* and *U* are stored in *pDst*or *ppDst*. Matrix elements located below the matrix diagonal are the lower triangular matrix *L*. The unit diagonal elements of the matrix *L* are not stored. The remaining matrix elements are the upper triangular matrix *U*. If necessary, the function implements the algorithm with partial pivoting that interchanges rows. Array of pivot indices is stored in *pDstIndex*.

LU Decomposition Matrix Storage



Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the size of the source matrix is equal to 0.
`ippStsSingularErr`	Returns an error when the source matrix is singular.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the *roiShift* value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the *count* value is less or equal to zero.

LUBackSubst

Solves system of linear equations with LU-factored square matrix.

Syntax

Case 1: Matrix - vector operation

```
IppStatus ippmLUBackSubst_mv_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2,
int* pSrcIndex, const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst, int dstStride2, int
widthHeight);
```

```
IppStatus ippmLUBackSubst_mv_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2,
int* pSrcIndex, const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst, int dstStride2, int
widthHeight);
```

```
IppStatus ippmLUBackSubst_mv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int* pSrcIndex,
const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int widthHeight);
```

```
IppStatus ippmLUBackSubst_mv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int* pSrcIndex,
const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int widthHeight);
```

Case 2: Matrix - vector array operation

```
IppStatus ippmLUBackSubst_mva_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1,
int src1Stride2, int* pSrcIndex, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2,
Ipp32f* pDst, int dstStride0, int dstStride2, int widthHeight, int count);

IppStatus ippmLUBackSubst_mva_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1,
int src1Stride2, int* pSrcIndex, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2,
Ipp64f* pDst, int dstStride0, int dstStride2, int widthHeight, int count);

IppStatus ippmLUBackSubst_mva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int* pSrcIndex,
const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst, int dstRoiShift,
int dstStride2, int widthHeight, int count);

IppStatus ippmLUBackSubst_mva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int* pSrcIndex,
const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst, int dstRoiShift,
int dstStride0, int widthHeight, int count);

IppStatus ippmLUBackSubst_mva_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride1, int src1Stride2, int* pSrcIndex, const Ipp32f** ppSrc2, int src2RoiShift,
int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int widthHeight, int
count);

IppStatus ippmLUBackSubst_mva_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride1, int src1Stride2, int* pSrcIndex, const Ipp64f** ppSrc2, int src2RoiShift,
int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int widthHeight, int
count);
```

Case 3: Matrix array - vector array operation

```
IppStatus ippmLUBackSubst_mava_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1,
int src1Stride2, int* pSrcIndex, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2,
Ipp32f* pDst, int dstStride0, int dstStride2, int widthHeight, int count);

IppStatus ippmLUBackSubst_mava_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1,
int src1Stride2, int* pSrcIndex, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2,
Ipp64f* pDst, int dstStride0, int dstStride2, int widthHeight, int count);

IppStatus ippmLUBackSubst_mava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride0, int* pSrcIndex, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
Ipp32f** ppDst, int dstRoiShift, int dstStride0, int widthHeight, int count);

IppStatus ippmLUBackSubst_mava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride0, int* pSrcIndex, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
Ipp64f** ppDst, int dstRoiShift, int dstStride0, int widthHeight, int count);

IppStatus ippmLUBackSubst_mava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride1, int src1Stride2, int* pSrcIndex, const Ipp32f** ppSrc2, int src2RoiShift,
int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int widthHeight, int
count);

IppStatus ippmLUBackSubst_mava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride1, int src1Stride2, int* pSrcIndex, const Ipp64f** ppSrc2, int src2RoiShift,
int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int widthHeight, int
count);
```

Parameters

pSrc1, ppSrc1	Pointer to the source matrix or array of matrices. Matrix(ces) must be a result of calling `LUDecomp`.
src1Stride0	Stride between the matrices in the source matrix array.
src1Stride1	Stride between the rows in the source matrix(ces).
src1Stride2	Stride between the elements in the source matrix(ces).
src1RoiShift	ROI shift in the source matrix(ces).
pSrc2, ppSrc2	Pointer to the source vector or array of vectors.
src2Stride0	Stride between the vectors in the source vector array.
src2Stride2	Stride between the elements in the source vector(s).
src2RoiShift	ROI shift in the source vector(s).
pSrcIndex	Pointer to array of pivot indices. This array must be a result of calling `LUDecomp`. The array size can be more than or equal to *widthHeight*. If the operation is performed on an array of matrices, the size of the array of indices must be more than or equal to *count *widthHeight*.
pDst, ppDst	Pointer to the destination vector or array of vectors.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector(s).
dstRoiShift	ROI shift in the destination vector(s).
widthHeight	Size of the square matrix, the source vector, and the destination vector.
count	Number of matrices and right-hand part vectors in the arrays.

Description

The function `ippmLUBackSubst` is declared in the `ippm.h` header file. The function solves for *x* the following systems of linear equations:

$$A \cdot x = L \cdot U \cdot x = b\ ,$$

where *A* is the matrix of linear equations system, stored in *pSrc1* or *ppSrc1*, *b* is the vector of the right-hand side, stored in *pSrc2* or *ppSrc2*, *x* is the unknown vector, stored in *pDst* or *ppDst*.

You should call the function `ippmLUDecomp` to compute the LU decomposition of *A* before calling `ippmLUBackSubst`.

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the size of the source matrix is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.

`ippStsRoiShiftMatrixErr`	Returns an error when the *roiShift* value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the *count* value is less or equal to zero.

Example

The code example below demonstrates how to use the functions `ippmLUDecomp_m_32f` and `ippmLUBackSubst_mva_32f`.

```
IppStatus LUFactorization_32f(void){
    /* Source matrix with widthHeight=3 */
    Ipp32f pSrc[3*3] = { 3, -5, -10,
                        -1,  4,  2 ,
                         1, -2, -3 };
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 3*sizeof(Ipp32f);

    /* Solver right-part is 2 vectors with length=3 */
    Ipp32f pSrc2[3*2] = { 0,  2, 1,
                          1, -1, 2 };
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride0 = 3*sizeof(Ipp32f);

    Ipp32f pDecomp[3*3]; /* Decomposed matrix location */
    int decompStride2 = sizeof(Ipp32f);
    int decompStride1 = 3*sizeof(Ipp32f);

    Ipp32f pDst[3*2];    /* Solver destination location */
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = 3*sizeof(Ipp32f);
    int pIndex[3];       /* Pivoting indeces location */
    int widthHeight = 3;
    int count = 2;

    IppStatus status = ippmLUDecomp_m_32f((const Ipp32f*)pSrc,
        srcStride1, srcStride2, pIndex,
        pDecomp, decompStride1, decompStride2, widthHeight);

    /*

    // It is required for LU decomposition function to check return status

    // for catching wrong result in case of invalid input data


    */

    if (status == ippStsNoErr) {

        status = ippmLUBackSubst_mva_32f((const Ipp32f*)pDecomp,
```

```
            decompStride1, decompStride2, pIndex, pSrc2, src2Stride0,
            src2Stride2, pDst, dstStride0, dstStride2, widthHeight, count);

        printf_m_Ipp32f("LUDecomp result:", pDecomp, 3, 3, status);
        printf_v_int("Pivoting indices:", pIndex, 3);
        printf_va_Ipp32f("2 destination vectors:", pDst, 3, 2, status);

    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }

    return status;
}
```

Output:

LUDecomp result:

```
3.000000  -5.000000  -10.000000
-0.333333  2.333333  -1.333333
0.333333  -0.142857  0.142857
Pivoting indices:
0  1  2
2 destination vectors:
40.000000  6.000000  9.000000
47.000000  6.000000  11.000000
```

CholeskyDecomp

Performs Cholesky decomposition of a symmetric positive definite square matrix.

Syntax

Case 1: Matrix operation

IppStatus ippmCholeskyDecomp_m_32f(const Ipp32f* *pSrc*, int *srcStride1*, int *srcStride2*, Ipp32f* *pDst*, int *dstStride1*, int *dstStride2*, int *widthHeight*);

IppStatus ippmCholeskyDecomp_m_64f(const Ipp64f* *pSrc*, int *srcStride1*, int *srcStride2*, Ipp64f* *pDst*, int *dstStride1*, int *dstStride2*, int *widthHeight*);

IppStatus ippmCholeskyDecomp_m_32f_P(const Ipp32f** *ppSrc*, int *srcRoiShift*, Ipp32f** *ppDst*, int *dstRoiShift*, int *widthHeight*);

```
IppStatus ippmCholeskyDecomp_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f** ppDst,
int dstRoiShift, int widthHeight);
```

Case 2: Matrix array operation

```
IppStatus ippmCholeskyDecomp_ma_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1,
int srcStride2, Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int
widthHeight, int count);

IppStatus ippmCholeskyDecomp_ma_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1,
int srcStride2, Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2, int
widthHeight, int count);

IppStatus ippmCholeskyDecomp_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride2,
Ipp32f** ppDst, int dstRoiShift, int dstStride2, int widthHeight, int count);

IppStatus ippmCholeskyDecomp_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride2,
Ipp64f** ppDst, int dstRoiShift, int dstStride2, int widthHeight, int count);

IppStatus ippmCholeskyDecomp_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1,
int srcStride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int
widthHeight, int count);

IppStatus ippmCholeskyDecomp_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1,
int srcStride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1, int dstStride2, int
widthHeight, int count);
```

Parameters

pSrc, *ppSrc*	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDst, *ppDst*	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
widthHeight	Size of the square matrix.
count	Number of matrices in the array.

Description

The function `ippmCholeskyDecomp` is declared in the `ippm.h` header file. The function performs Cholesky decomposition of the symmetric square matrix that is positive definite. The source matrix is represented as a product of two matrices L and L^T, where L is the lower triangular and L^T is its transpose that can serve itself as the upper triangular. Result of the function operation ismatrix L with inverse diagonal elements. The function uses only data in the lower triangular of the source matrix.

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsNotPosDefErr	Returns an error when the source matrix is not positive definite.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the *roiShift* value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the *count* value is less or equal to zero.

CholeskyBackSubst

Solves system of linear equations using the Cholesky triangular factor.

Syntax

Case 1: Matrix -vector operation

```
IppStatus ippmCholeskyBackSubst_mv_32f(const Ipp32f* pSrc1, int src1Stride1, int
src1Stride2, const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst, int dstStride2, int
widthHeight);
```

```
IppStatus ippmCholeskyBackSubst_mv_64f(const Ipp64f* pSrc1, int src1Stride1, int
src1Stride2, const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst, int dstStride2, int
widthHeight);
```

```
IppStatus ippmCholeskyBackSubst_mv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, const
Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int widthHeight);
```

```
IppStatus ippmCholeskyBackSubst_mv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, const
Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int widthHeight);
```

Case 2: Matrix - vector array operation

```
IppStatus ippmCholeskyBackSubst_mva_32f(const Ipp32f* pSrc1, int src1Stride1, int
src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2, Ipp32f* pDst, int
dstStride0, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmCholeskyBackSubst_mva_64f(const Ipp64f* pSrc1, int src1Stride1, int
src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2, Ipp64f* pDst, int
dstStride0, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmCholeskyBackSubst_mva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, const
Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int
dstStride2, int widthHeight, int count);
```

```
IppStatus ippmCholeskyBackSubst_mva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, const
Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int
dstStride2, int widthHeight, int count);
```

```
IppStatus ippmCholeskyBackSubst_mva_32f_L(const Ipp32f* pSrc1, int src1Stride1, int
src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp32f** ppDst,
int dstRoiShift, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmCholeskyBackSubst_mva_64f_L(const Ipp64f* pSrc1, int src1Stride1, int
src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp64f** ppDst,
int dstRoiShift, int dstStride2, int widthHeight, int count);
```

Case 3: Matrix array - vector array operation

```
IppStatus ippmCholeskyBackSubst_mava_32f(const Ipp32f* pSrc1, int src1Stride0, int
src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2,
Ipp32f* pDst, int dstStride0, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmCholeskyBackSubst_mava_64f(const Ipp64f* pSrc1, int src1Stride0, int
src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2,
Ipp64f* pDst, int dstStride0, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmCholeskyBackSubst_mava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp32f** ppDst,
int dstRoiShift, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmCholeskyBackSubst_mava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp64f** ppDst,
int dstRoiShift, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmCholeskyBackSubst_mava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride1, int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2,
Ipp32f** ppDst, int dstRoiShift, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmCholeskyBackSubst_mava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride1, int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2,
Ipp64f** ppDst, int dstRoiShift, int dstStride2, int widthHeight, int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the source matrix or array of matrices. Must be a result of calling CholeskyDecomp.
src1Stride0	Stride between the matrices in the source array.
src1Stride1	Stride between the rows in the source matrix(ces).
src1Stride2	Stride between the elements in the source matrix(ces).
src1RoiShift	ROI shift in the source matrix(ces).
pSrc2, ppSrc2	Pointer to the source vector or array of vectors.
src2Stride0	Stride between the vectors in the source array.
src2Stride1	Stride between the elements in the source vector(s).
src2RoiShift	ROI shift in the source vector(s).
pDst, ppDst	Pointer to the destination vector or array of vectors.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector(s).
dstRoiShift	ROI shift in the destination vector(s).

widthHeight	Size of the square matrix, the source vector, and the destination vector.
count	Number of matrices and right-hand part vectors in the arrays.

Description

The function `ippmCholeskyBackSubst` is declared in the `ippm.h` header file. The function solves for *x* the following systems of linear equations:

$$A \cdot x = L \cdot L^T \cdot x = b \,,$$

where *A* is the matrix of linear equations system, stored in *pSrc1* or *ppSrc1*, *b* is the vector of the right-hand side, stored in *pSrc2* or *ppSrc2*, *x* is the unknown vector, stored in *pDst* or *ppDst*.

You should call the function ippmCholeskyDecomp to perform Cholesky decomposition of *A* before calling `ippm-CholeskyBackSubst`.

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the size of the source matrix is 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the *roiShift* value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the *count* value is less or equal to zero.

Example

The code example below demonstrates how to use the functions `ippmCholeskyDecomp_m_32f` and `ippmCholeskyBackSubst_mva_32f`.

```
IppStatus cholesky_mva_32f(void){
    /* Source matrix with widthHeight=4 */
    Ipp32f pSrc[4*4] = { 10, 1,  2, 3,
                          1, 12,  4, 5,
                          2,  4, 13, 6,
                          3,  5,  6, 14 };

    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 4*sizeof(Ipp32f);

    /* Solver right-part is 3 vectors with length=4 */
    Ipp32f pSrc2[3*4] = {1, 2,  3,  4,
                          5, 6,  7,  8,
                          9, 10, 11, 12 };
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride0 = 4*sizeof(Ipp32f);

    Ipp32f pDecomp[4*4] = {0}; /* Decomposed matrix location */
    int decompStride2 = sizeof(Ipp32f);
    int decompStride1 = 4*sizeof(Ipp32f);
```

```
    Ipp32f pDst[3*4];          /* Solver destination location */
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = 4*sizeof(Ipp32f);

    int widthHeight = 4;
    int count = 3;


    IppStatus status = ippmCholeskyDecomp_m_32f((const Ipp32f*)pSrc,
        srcStride1, srcStride2, pDecomp, decompStride1,
        decompStride2, widthHeight);

    /*

    // It is required for Cholesky decomposition function to check return status


    // for catching wrong result in case of invalid input data


    */

    if (status == ippStsNoErr) {
        status = ippmCholeskyBackSubst_mva_32f((const Ipp32f*)pDecomp,
            decompStride1, decompStride2, pSrc2, src2Stride0, src2Stride2,
            pDst, dstStride0, dstStride2, widthHeight, count);

        printf_m_Ipp32f("Cholesky decomposition:", pDecomp, 4, 4, status);
        printf_va_Ipp32f("3 destination vectors:", pDst, 4, 3, status);

    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }

    return status;
}
```

Output:

```
Cholesky decomposition:

0.316228 0.000000 0.000000 0.000000

0.316228 0.289886 0.000000 0.000000

0.632456 1.101565 0.296349 0.000000

0.948683 1.362462 1.155513 0.317685

3 destination vectors:

0.006629 0.034783 0.116639 0.221883

0.332266 0.260316 0.273350 0.290109

0.657903 0.485848 0.430061 0.358335
```

Least Squares Problem Functions 7

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) functions for small matrices that compute the matrix QR decomposition and solve the least squares (LS) problem to an overdetermined system of linear equations. A typical least-squares problem is as follows: given a matrix *A* and a vector *b*, find the vector *x* that minimizes the L2-norm

$$\|Ax - b\|^2.$$

The number of rows in matrix *A* is equal to *`height`* and the number of columns is equal to *`width`*, rank (*A*) = *`width`*.

Least Squares Problem functions

Function Base Name	Operation
`QRDecomp`	Computes the QR decomposition for the given matrix.
`QRBackSubst`	Solves least squares problem for QR-decomposed matrix.

QRDecomp

Computes the QR decomposition for the given matrix.

Syntax

Case 1: Matrix operation

```
IppStatus ippmQRDecomp_m_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2, Ipp32f* pBuffer, Ipp32f* pDst, int dstStride1, int dstStride2, int width, int height);
```

```
IppStatus ippmQRDecomp_m_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2, Ipp64f* pBuffer, Ipp64f* pDst, int dstStride1, int dstStride2, int width, int height);
```

```
IppStatus ippmQRDecomp_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f* pBuffer, Ipp32f** ppDst, int dstRoiShift, int width, int height);
```

```
IppStatus ippmQRDecomp_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f* pBuffer, Ipp64f** ppDst, int dstRoiShift, int width, int height);
```

Case 2: Matrix array operation

```
IppStatus ippmQRDecomp_ma_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1, int srcStride2, Ipp32f* pBuffer, Ipp32f* pDst, int dstStride0, int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmQRDecomp_ma_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1, int
srcStride2, Ipp64f* pBuffer, Ipp64f* pDst, int dstStride0, int dstStride1, int dstStride2,
int width, int height, int count);
```

```
IppStatus ippmQRDecomp_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift, int srcStride0,
Ipp32f* pBuffer, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
int count);
```

```
IppStatus ippmQRDecomp_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift, int srcStride0,
Ipp64f* pBuffer, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
int count);
```

```
IppStatus ippmQRDecomp_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift, int srcStride1,
int srcStride2, Ipp32f* pBuffer, Ipp32f** ppDst, int dstRoiShift, int dstStride1, int
dstStride2, int width, int height, int count);
```

```
IppStatus ippmQRDecomp_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift, int srcStride1,
int srcStride2, Ipp64f* pBuffer, Ipp64f** ppDst, int dstRoiShift, int dstStride1, int
dstStride2, int width, int height, int count);
```

Parameters

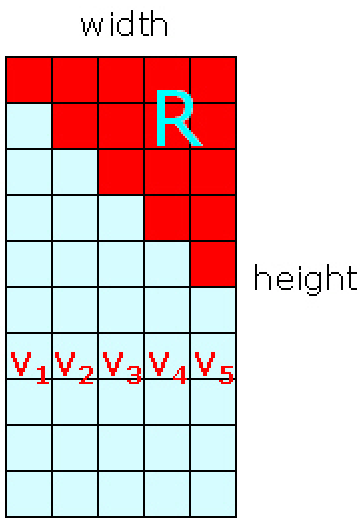
pSrc, *ppSrc*	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pBuffer	Pointer to a pre-allocated auxiliary array to be used for internal computations. The number of elements in the array must be at least *height*.
pDst, *ppDst*	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
width	Matrix width.
height	Matrix height.
count	Number of matrices in the array.

Description

The function `ippmQRDecomp` is declared in the `ippm.h` header file. The function computes the QR decomposition of a general matrix *A* without the use of pivoting. The number of rows *height* is greater or equal to the number of columns *width*, that is rank (*A*) = *width*.

The function forms the matrix Q implicitly. Instead of Q, Householder vectors V_n are stored, *width* in number. The first (n -1) components of each Householder vector V_n are not stored because they are equal to 0. The n-th component of each Householder vector V_n is not stored either because it is equal to 1. The last (*height* - n) elements of vector V_n are located below the diagonal of the *pDst*, or *ppDst*, matrix. The remaining elements of the *pDst*, or *ppDst*, matrix form the upper triangular matrix R (see Figure QR Decomposition Matrix Storage).

QR Decomposition Matrix Storage



Mathematically, the matrix Q is represented as a product of *width* elementary reflections

$Q = H_1 * H_2 *...* H_k$, where k = *width*.

Each elementary reflection is

$H_n = I - (2/r) V_n * V_n^T$,

where V_n is the Householder vector, I is the identity matrix,

$$r = \|V_n\|_2^2, n = 1, \dots ,width.$$

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when one of the input pointers is `NULL`.
`ippStsSizeErr`	Returns an error when the size of the source matrix is equal to 0.
`ippStsDivByZeroErr`	Returns an error when the source matrix has an incomplete column rank.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.

`ippStsRoiShiftMatrixErr`	Returns an error when the *roiShift* value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.
`ippStsSizeMatchMatrixErr`	Returns an error when the sizes of the source matrices are unsuitable.

QRBackSubst

Solves least squares problem for QR-decomposed matrix.

Syntax

Case 1: Matrix - vector operation

```
IppStatus ippmQRBackSubst_mv_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2,
Ipp32f* pBuffer, const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst, int dstStride2, int
width, int height);
```

```
IppStatus ippmQRBackSubst_mv_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2,
Ipp64f* pBuffer, const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst, int dstStride2, int
width, int height);
```

```
IppStatus ippmQRBackSubst_mv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, Ipp32f* pBuffer,
const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift, int width, int
height);
```

```
IppStatus ippmQRBackSubst_mv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, Ipp64f* pBuffer,
const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift, int width, int
height);
```

Case 2: Matrix - vector array operation

```
IppStatus ippmQRBackSubst_mva_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1,
int src1Stride2, Ipp32f* pBuffer, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2,
Ipp32f* pDst, int dstStride0, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmQRBackSubst_mva_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1,
int src1Stride2, Ipp64f* pBuffer, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2,
Ipp64f* pDst, int dstStride0, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmQRBackSubst_mva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, Ipp32f*
pBuffer, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst, int
dstRoiShift, int dstStride0, int width, int height, int count);
```

```
IppStatus ippmQRBackSubst_mva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, Ipp64f*
pBuffer, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst, int
dstRoiShift, int dstStride0, int width, int height, int count);
```

```
IppStatus ippmQRBackSubst_mva_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride1, int src1Stride2, Ipp32f* pBuffer, const Ipp32f** ppSrc2, int src2RoiShift,
int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int width, int height,
int count);
```

```
IppStatus ippmQRBackSubst_mva_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride1, int src1Stride2, Ipp64f* pBuffer, const Ipp64f** ppSrc2, int src2RoiShift,
int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int width, int height,
int count);
```

Case 3: Matrix array - vector array operation

```
IppStatus ippmQRBackSubst_mava_32f(const Ipp32f* pSrc1, int src1Stride0, int src1Stride1,
int src1Stride2, Ipp32f* pBuffer, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2,
Ipp32f* pDst, int dstStride0, int dstStride2, int width, int height, int count);

IppStatus ippmQRBackSubst_mava_64f(const Ipp64f* pSrc1, int src1Stride0, int src1Stride1,
int src1Stride2, Ipp64f* pBuffer, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2,
Ipp64f* pDst, int dstStride0, int dstStride2, int width, int height, int count);

IppStatus ippmQRBackSubst_mava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride0, Ipp32f* pBuffer, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmQRBackSubst_mava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride0, Ipp64f* pBuffer, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmQRBackSubst_mava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift, int
src1Stride1, int src1Stride2, Ipp32f* pBuffer, const Ipp32f** ppSrc2, int src2RoiShift,
int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int width, int height,
int count);

IppStatus ippmQRBackSubst_mava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift, int
src1Stride1, int src1Stride2, Ipp64f* pBuffer, const Ipp64f** ppSrc2, int src2RoiShift,
int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int width, int height,
int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the source matrix or array of matrices. The matrix(ces) must be a result of calling ippmQRDecomp .
src1Stride0	Stride between the matrices in the source matrix array.
src1Stride1	Stride between the rows in the source matrix(ces).
src1Stride2	Stride between the elements in the source matrix(ces).
src1RoiShift	ROI shift in the source matrix(ces).
pSrc2, ppSrc2	Pointer to the source vector or array of vectors.
src2Stride0	Stride between the vectors in the source vector array.
src2Stride2	Stride between the elements in the source vector(s).
src2RoiShift	ROI shift in the source vector(s).
pBuffer	Pointer to a pre-allocated auxiliary array to be used for internal computations. The number of elements in the array must be at least *height*.
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the vectors in the destination array.

dstStride2	Stride between the elements of the destination vector(s).
dstRoiShift	ROI shift in the destination vector(s).
width	Matrix width and destination vector length.
height	Matrix height and source vector length.
count	Number of matrices and right-hand part vectors in the array.

Description

The function *ippmQRBackSubst* is declared in the ippm.h header file. The function solves the least squares problem for an overdetermined system of linear equations

$Ax = b$,

where

A is the matrix of linear equations system, stored in *pSrc1* or *ppSrc1*,

b is the vector of the right-hand side, stored in *pSrc2* or *ppSrc2*,

x is the unknown vector, stored in *pDst* or *ppDst*.

The number of equations in the system *height* is equal to or greater than the number of unknown variables *width*, *rank* (*A*) = *width*.

A typical least squares problem is as follows: given a matrix A and a vector b, find the vector x that minimizes the L2-norm

$$\|Ax - b\|^2.$$

You should call the function ippmQRDecomp to compute the QR decomposition of A before calling the ippmQRBackSubst function (See description for the function ippmQRDecomp.)

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when one of the input pointers is NULL.
ippStsSizeErr	Returns an error when the size of the source matrix is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	RoiShift value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.
ippStsSizeMatchMatrixErr	Returns an error when the sizes of the source matrices are unsuitable.

Example

The code example below demonstrates how to use the functions `ippmQRDecomp_m_32f` and `ippmQRBackSubst_mva_32f`.

```
IppStatus QRFactorization_32f(void) {
    /* Source matrix with width=4 and height=5 */
    Ipp32f pSrc[5*4] = {1, 1, 1, 1,
                        1, 3, 1, 1,
                        1,-1, 3, 1,
                        1, 1, 1, 3,
                        1, 1, 1, -1 };
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 4*sizeof(Ipp32f);

        /* Solver right-part is 2 vectors with length=5 */
    Ipp32f pSrc2[2*5] = { 2, 1, 6, 3, 1,
                          3, 4, 5, 6, 1 };
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride0 = 5*sizeof(Ipp32f);
    Ipp32f pDecomp[5*4]; /* Decomposed matrix location */
    int decompStride2 = sizeof(Ipp32f);
    int decompStride1 = 4*sizeof(Ipp32f);

    Ipp32f pDst[2*4]; /* Solver destination location */
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = 4*sizeof(Ipp32f);

    int width  = 4;
    int height = 5;
    int count  = 2;

    Ipp32f pBuffer[5];  /* Buffer location */

    IppStatus status = ippmQRDecomp_m_32f((const Ipp32f*)pSrc,
```

```
        srcStride1, srcStride2, pBuffer,
        pDecomp, decompStride1, decompStride2, width, height);

    status = ippmQRBackSubst_mva_32f((const Ipp32f*)pDecomp,
       decompStride1, decompStride2, pBuffer, pSrc2, src2Stride0,
       src2Stride2, pDst, dstStride0, dstStride2, width, height, count);

    /*
    // It is required for QR decomposition function to check return status
    // for catching wrong result in case of invalid input data
    */
    if (status == ippStsNoErr) {
       status = ippmQRBackSubst_mva_32f((const Ipp32f*)pDecomp,
           decompStride1, decompStride2, pBuffer, pSrc2, src2Stride0,
           src2Stride2, pDst, dstStride0, dstStride2, width, height, count);



        printf_m_Ipp32f("QRDecomp result:", pDecomp, 4, 5, status);
        printf_va_Ipp32f("2 destination vectors:", pDst, 4, 2, status);

    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
    return status;
}
```

Output:

```
QRDecomp result:
-2.236068 -2.236068  -3.130495  -2.236068
0.309017  -2.828427   1.414214  -0.000000
0.309017  -0.414214  -1.095445  0.000000
0.309017  -0.000000  -0.130449  -2.828427
0.309017  -0.000000  -0.130449  -0.414214
2 destination vectors:
```

```
0.500000  -0.500000   1.500000   0.500000
0.583333  0.333333     1.166667    1.250000
```

Eigenvalue Problem Functions

8

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) functions for small matrices that compute eigenvalues and eigenvectors for real symmetric and general (nonsymmetric) square matrices.

Eigenvalue Problem Functions

Function Base Name	Operation
`EigenValuesVectorsSym`	Finds eigenvalues and eigenvectors for real symmetric matrices (solves symmetric eigenvalue problem).
`EigenValuesSym`	Finds eigenvalues for real symmetric matrices.
`EigenValuesVectors`	Finds eigenvalues, right and left eigenvectors for real general (nonsymmetric) matrices (solves nonsymmetric eigenvalue problem).
`EigenValues`	Finds eigenvalues for real general (nonsymmetric) matrices.
`EigenValuesVectorsGetBufSize`	Computes the work buffer size for the functions `EigenValuesVectors`.
`EigenValuesGetBufSize`	Computes the work buffer size for the functions `EigenValues`.

EigenValuesVectorsSym

Finds eigenvalues and eigenvectors for real symmetric matrices (solves symmetric eigenvalue problem).

Syntax

Case 1: Matrix operation

```
IppStatus ippmEigenValuesVectorsSym_m_32f (const Ipp32f* pSrc, int srcStride1, int
srcStride2, Ipp32f* pBuffer, Ipp32f* pDstVectors, int dstStride1, int dstStride2, Ipp32f*
pDstValues, int widthHeight);
```

```
IppStatus ippmEigenValuesVectorsSym_m_64f (const Ipp64f* pSrc, int srcStride1, int
srcStride2, Ipp64f* pBuffer, Ipp64f* pDstVectors, int dstStride1, int dstStride2, Ipp64f*
pDstValues, int widthHeight);
```

```
IppStatus ippmEigenValuesVectorsSym_m_32f_P (const Ipp32f** ppSrc, int srcRoiShift, Ipp32f*
pBuffer, Ipp32f** ppDstVectors, int dstRoiShift, Ipp32f* pDstValues, int widthHeight);
```

IppStatus ippmEigenValuesVectorsSym_m_64f_P (const Ipp64f** *ppSrc*, int *srcRoiShift*, Ipp64f* *pBuffer*, Ipp64f** *ppDstVectors*, int *dstRoiShift*, Ipp64f* *pDstValues*, int *widthHeight*);

Case 2: Matrix array operation

IppStatus ippmEigenValuesVectorsSym_ma_32f (const Ipp32f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp32f* *pBuffer*, Ipp32f* *pDstVectors*, int *dstStride0*, int *dstStride1*, int *dstStride2*, Ipp32f* *pDstValues*, int *widthHeight*, int *count*);

IppStatus ippmEigenValuesVectorsSym_ma_32f_P (const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, Ipp32f* *pBuffer*, Ipp32f** *ppDstVectors*, int *dstRoiShift*, int *dstStride0*, Ipp32f* *pDstValues*, int *widthHeight*, int *count*);

IppStatus ippmEigenValuesVectorsSym_ma_32f_L (const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride1*, int *srcStride2*, Ipp32f* *pBuffer*, Ipp32f** *ppDstVectors*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, Ipp32f* *pDstValues*, int *widthHeight*, int *count*);

IppStatus ippmEigenValuesVectorsSym_ma_64f (const Ipp64f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp64f* *pBuffer*, Ipp64f* *pDstVectors*, int *dstStride0*, int *dstStride1*, int *dstStride2*, Ipp64f* *pDstValues*, int *widthHeight*, int *count*);

IppStatus ippmEigenValuesVectorsSym_ma_64f_P (const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, Ipp64f* *pBuffer*, Ipp64f** *ppDstVectors*, int *dstRoiShift*, int *dstStride0*, Ipp64f* *pDstValues*, int *widthHeight*, int *count*);

IppStatus ippmEigenValuesVectorsSym_ma_64f_L (const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride1*, int *srcStride2*, Ipp64f* *pBuffer*, Ipp64f** *ppDstVectors*, int *dstRoiShift*, int *dstStride1*, int *dstStride2*, Ipp64f* *pDstValues*, int *widthHeight*, int *count*);

Parameters

pSrc, *ppSrc*	Pointer to the source matrix or array of matrices.
srcStride0	Stride between matrices in the source array.
srcStride1	Stride between rows in the source matrix(ces).
srcStride2	Stride between elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pBuffer	Pointer to a pre-allocated auxiliary array to be used for internal computations. The number of elements in the array must be at least $widthHeight^2$.
pDstVectors, *ppDstVectors*	Pointer to the destination matrix or array of matrices whose columns are eigenvectors.
dstStride0	Stride between matrices in the destination array.
dstStride1	Stride between rows in the destination matrix.
dstStride2	Stride between elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
pDstValues	Pointer to the destination dense array that contains eigenvalues. The number of elementes in the array must be at least *widthHeight* for a matrix and *widthHeight*count* for an array of matrices.

widthHeight	Size of the square matrix.
count	The number of matrices in the array.

Description

The function `ippmEigenValuesVectorsSym` is declared in the `ippm.h` header file.

The function solves the Symmetric Eigenvalue Problem, that is finds the eigenvalues λ and corresponding eigenvectors $z \neq 0$ such that

$$Az = \lambda z ,$$

where *A* is a real symmetric square matrix of size *widthHeight*.

In case of a real symmetric matrix, all the *widthHeight* eigenvalues are real and there exists an orthonormal system of *widthHeight* eigenvectors. When all eigenvalues and eigenvectors are computed, the classical spectral factorization of *A* is

$$A = Z \Lambda Z^T ,$$

where Λ is a diagonal matrix whose non-zero elements are the eigenvalues, *Z* is an orthogonal matrix whose columns are the eigenvectors, and Z^T is its transpose.

The function stores eigenvalues in the array pointed by *pDstValues* densely and in the decreasing order. Eigenvectors of a source matrix are placed in columns of the appropriate destination matrix, pointed by *pDstVectors* or *ppDstVectors*.

The function uses only data in the lower triangular part of a source matrix **pSrc* or **ppSrc*.

Return Values

`ippStsOk`	Indicates no errors.
`ippStsNullPtrErr`	Indicates an error if at least one input pointer is NULL.
`ippStsSizeErr`	Indicates an error if the input size parameter is less or equal to 0.
`ippStsStrideMatrixErr`	Indicates an error if a stride value is not positive or not divisible by the size of data type.
`ippStsRoiShiftMatrixErr`	Indicates an error if a roiShift value is negative or not divisible by the size of data type.
`ippStsCountMatrixErr`	Indicates an error when the count value is less or equal to 0.
`ippStsSingularErr`	Indicates an error if any of the input matrices is singular.
`ippStsConvergeErr`	Indicates an error if the algorithm does not converge.

Example

The code example below demonstrates how to use the function `ippmEigenValuesVectorsSym_m_32f`.

```
IppStatus eigen_problem_32fvoid){
    /* Source matrix with width=4 and height=4 */

    Ipp32f pSrc[4*4]= {1, 1, 1, 3,
                       1, 3, 1, 3,
                       1, 1, 3, 1,
                       3, 3, 1, 3};

    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 4*sizeof(Ipp32f);

    Ipp32f pBuffer[4*4]; /* Buffer location */
    int widthHeight = 4; Ipp32f pDstValues[4]; /* Eigenvalues location */

    Ipp32f pDstVectors[4*4]; /* Eigenvectors location */


    int dstStride2 = sizeof(Ipp32f); int dstStride1 = 4*sizeof(Ipp32f);
    IppStatus status=ippmEigenValuesVectorsSym_m_32f((const Ipp32f*)pSrc,
    srcStride1, srcStride2, pBuffer,
    pDstVectors, dstStride1, dstStride2, pDstValues, widthHeight);


    /*
    // It is required for EigenValuesVectors function to check return status
    // for catching wrong result in case of invalid input data
    */
    if (status == ippStsNoErr) {
       printf_va_Ipp32f("Eigenvalues:", pDstValues, 4, 1, status);
       printf_m_Ipp32f("Eigenvectors :", pDstVectors, 4, 4, status);
    } else {
     printf("Function returns status: %s \n", ippGetStatusString(status));
    }

   return status;
}
```

Output:

```
Eigenvalues:

7.911653  2.443112  1.121464  -1.476230

Eigenvectors:

-0.409522  -0.040703  -0.587074  -0.697122

-0.548069  -0.224421  0.749409   -0.296043

-0.327626  0.938908   0.071989   0.077017

-0.651593  -0.257742  -0.297569  0.648420
```

EigenValuesSym

Finds eigenvalues for real symmetric matrices.

Syntax

Case 1: Matrix operation

```
IppStatus ippmEigenValuesSym_m_32f (const Ipp32f* pSrc, int srcStride1, int srcStride2,
Ipp32f* pBuffer, Ipp32f* pDstValues, int widthHeight);
```

```
IppStatus ippmEigenValuesSym_m_64f (const Ipp64f* pSrc, int srcStride1, int srcStride2,
Ipp64f* pBuffer, Ipp64f* pDstValues, int widthHeight);
```

```
IppStatus ippmEigenValuesSym_m_32f_P (const Ipp32f** ppSrc, int srcRoiShift,  Ipp32f*
pBuffer, Ipp32f* pDstValues, int widthHeight);
```

```
IppStatus ippmEigenValuesSym_m_64f_P (const Ipp64f** ppSrc, int srcRoiShift, Ipp64f*
pBuffer, Ipp64f* pDstValues, int widthHeight);
```

Case 2: Matrix array operation

```
IppStatus ippmEigenValuesSym_ma_32f (const Ipp32f* pSrc, int srcStride0, int srcStride1,
int srcStride2, Ipp32f* pBuffer, Ipp32f* pDstValues, int widthHeight, int count);
```

```
IppStatus ippmEigenValuesSym_ma_32f_P (const Ipp32f** ppSrc, int srcRoiShift,  int
srcStride0, Ipp32f* pBuffer, Ipp32f* pDstValues, int widthHeight, int count);
```

```
IppStatus ippmEigenValuesSym_ma_32f_L (const Ipp32f** ppSrc, int srcRoiShift, int
srcStride1, int srcStride2, Ipp32f* pBuffer, Ipp32f* pDstValues, int widthHeight, int
count);
```

```
IppStatus ippmEigenValuesSym_ma_64f (const Ipp64f* pSrc, int srcStride0, int srcStride1,
int srcStride2, Ipp64f* pBuffer, Ipp64f* pDstValues, int widthHeight, int count);
```

```
IppStatus ippmEigenValuesSym_ma_64f_P (const Ipp64f** ppSrc, int srcRoiShift, int
srcStride0, Ipp64f* pBuffer, Ipp64f* pDstValues, int widthHeight, int count);
```

```
IppStatus ippmEigenValuesSym_ma_64f_L (const Ipp64f** ppSrc, int srcRoiShift, int srcStride1,
int srcStride2, Ipp64f* pBuffer, Ipp64f* pDstValues, int widthHeight, int count );
```

Parameters

pSrc, *ppSrc*	Pointer to the source matrix or array of matrices.
srcStride0	Stride between matrices in the source array.
srcStride1	Stride between rows in the source matrix(ces).
srcStride2	Stride between elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pBuffer	Pointer to a pre-allocated auxiliary array to be used for internal computations. The number of elements in the array must be at least *widthHeight*

pDstValues	Pointer to the destination dense array that contains eigenvalues. The number of elements in the array must be at least *widthHeight* for a matrix and *widthHeight*count* for an array of matrices.
widthHeight	Size of the square matrix.
count	The number of matrices in the array.

Description

The function `ippmEigenValuesSym` is declared in the `ippm.h` header file.

The function finds eigenvalues for a real symmetric matrix *A* of size *widthHeight*. Each eigenvalue λ is a scalar such that

$Az=\lambda z$,

for a vector $z \neq 0$ called an eigenvector. In case of a real symmetric matrix, all the *widthHeight* eigenvalues are real. The function stores eigenvalues in the array pointed by *pDstValues* densely and in the decreasing order.

The function uses only data in the lower triangular part of a source matrix **pSrc* or **ppSrc*.

Return Values

ippStsOk	Indicates no errors.
ippStsNullPtrErr	Indicates an error if at least one input pointer is NULL.
ippStsSizeErr	Indicates an error if the input size parameter is less or equal to 0.
ippStsStrideMatrixErr	Indicates an error if a stride value is not positive or not divisible by the size of data type.
ippStsRoiShiftMatrixErr	Indicates an error if a *roiShift* value is negative or not divisible by the size of data type.
ippStsCountMatrixErr	Indicates an error when the count value is less or equal to 0.
ippStsSingularErr	Indicates an error if any of the input matrices is singular.
ippStsConvergeErr	Indicates an error if the algorithm does not converge.

Example

The code example below demonstrates how to use the function `ippmEigenValuesSym_ma_32f`.

```
IppStatus eigenvalues_ma_32f(void){
    /* Source data: 2 matrices with width=3 and height=3 */
    Ipp32f pSrc[2*3*3]= {1, 1, 1,
                         1, 3, 1,
                         1, 1, 3,
                         1, 1, 3,
                         1, 2, 1,
                         3, 1, 3};
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 3*sizeof(Ipp32f);
    int srcStride0 = 3*3*sizeof(Ipp32f);

    Ipp32f pBuffer[3*3]; /* Buffer location */
    int widthHeight = 3; int count = 2;
```

```
    Ipp32f pDstValues[2*3]; /* Eigenvalues location for two matrices */

    IppStatus status=ippmEigenValuesSym_ma_32f((const Ipp32f*)pSrc,
        srcStride0, srcStride1, srcStride2, pBuffer,
        pDstValues, widthHeight, count);

    /*
    // It is required for EigenValues function to check return status
    // for catching wrong result in case of invalid input data
    */
    if (status == ippStsNoErr) {
        printf_va_Ipp32f("Eigenvalues:", pDstValues, 3, 2, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
    return status;
}
```

Output:

```
Eigenvalues:

4.561553 2.000000 0.438447

5.691268 1.488873 -1.180140
```

EigenValuesVectors

Finds eigenvalues, right and left eigenvectors for real general (nonsymmetric) matrices (solves nonsymmetric eigenvalue problem).

Syntax

Case 1: Eigenvalues, right and left eigenvectors for a matrix

`IppStatus ippmEigenValuesVectors_m_32f (const Ipp32f* *pSrc*, int *srcStride1*, int *srcStride2*, Ipp32f* *pDstVectorsRight*, int *dstRightStride1*, int *dstRightStride2*, Ipp32f* *pDstVectorsLeft*, int *dstLeftStride1*, int *dstLeftStride2*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);`

`IppStatus ippmEigenValuesVectors_m_64f (const Ipp64f* *pSrc*, int *srcStride1*, int *srcStride2*, Ipp64f* *pDstVectorsRight*, int *dstRightStride1*, int *dstRightStride2*, Ipp64f* *pDstVectorsLeft*, int *dstLeftStride1*, int *dstLeftStride2*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);`

`IppStatus ippmEigenValuesVectors_m_32f_P (const Ipp32f** *ppSrc*, int *srcRoiShift*, Ipp32f** *ppDstVectorsRight*, int *dstRightRoiShift*, Ipp32f** *ppDstVectorsLeft*, int *dstLeftRoiShift*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);`

`IppStatus ippmEigenValuesVectors_m_64f_P (const Ipp64f** *ppSrc*, int *srcRoiShift*, Ipp64f** *ppDstVectorsRight*, int *dstRightRoiShift*, Ipp64f** *ppDstVectorsLeft*, int *dstLeftRoiShift*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);`

Case 2: Eigenvalues and right eigenvectors for a matrix

IppStatus ippmEigenValuesVectorsRight_m_32f (const Ipp32f* *pSrc*, int *srcStride1*, int *srcStride2*, Ipp32f* *pDstVectorsRight*, int *dstRightStride1*, int *dstRightStride2*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectorsRight_m_64f (const Ipp64f* *pSrc*, int *srcStride1*, int *srcStride2*, Ipp64f* *pDstVectorsRight*, int *dstRightStride1*, int *dstRightStride2*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectorsRight_m_32f_P (const Ipp32f** *ppSrc*, int *srcRoiShift*, Ipp32f** *ppDstVectorsRight*, int *dstRightRoiShift*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectorsRight_m_64f_P (const Ipp64f** *ppSrc*, int *srcRoiShift*, Ipp64f** *ppDstVectorsRight*, int *dstRightRoiShift*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);

Case 3: Eigenvalues and left eigenvectors for a matrix

IppStatus ippmEigenValuesVectorsLeft_m_32f (const Ipp32f* *pSrc*, int *srcStride1*, int *srcStride2*, Ipp32f* *pDstVectorsLeft*, int *dstLeftStride1*, int *dstLeftStride2*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectorsLeft_m_64f (const Ipp64f* *pSrc*, int srcStride1, int *srcStride2*, Ipp64f* *pDstVectorsLeft*, int *dstLeftStride1*, int *dstLeftStride2*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectorsLeft_m_32f_P (const Ipp32f** *ppSrc*, int *srcRoiShift*, Ipp32f** *ppDstVectorsLeft*, int *dstLeftRoiShift*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectorsLeft_m_64f_P (const Ipp64f** *ppSrc*, int *srcRoiShift*, Ipp64f** *ppDstVectorsLeft*, int *dstLeftRoiShift*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);

Case 4: Eigenvalues, right and left eigenvectors for a matrix array

IppStatus ippmEigenValuesVectors_ma_32f (const Ipp32f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp32f* *pDstVectorsRight*, int *dstRightStride0*, int *dstRightStride1*, int *dstRightStride2*, Ipp32f* *pDstVectorsLeft*, int *dstLeftStride0*, int *dstLeftStride1*, int *dstLeftStride2*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectors_ma_64f (const Ipp64f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp64f* *pDstVectorsRight*, int *dstRightStride0*, int *dstRightStride1*, int *dstRightStride2*, Ipp64f* *pDstVectorsLeft*, int *dstLeftStride0*, int *dstLeftStride1*, int *dstLeftStride2*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectors_ma_32f_P (const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, Ipp32f** *ppDstVectorsRight*, int *dstRightRoiShift*, int *dstRightStride0*, Ipp32f** *ppDstVectorsLeft*, int *dstLeftRoiShift*, int *dstLeftStride0*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectors_ma_64f_P (const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, Ipp64f** *ppDstVectorsRight*, int *dstRightRoiShift*, int *dstRightStride0*, Ipp64f** *ppDstVectorsLeft*, int *dstLeftRoiShift*, int *dstLeftStride0*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectors_ma_32f_L (const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride1*, int *srcStride2*, Ipp32f** *ppDstVectorsRight*, int *dstRightRoiShift*, int *dstRightStride1*, int *dstRightStride2*, Ipp32f** *ppDstVectorsLeft*, int *dstLeftRoiShift*, int *dstLeftStride1*, int *dstLeftStride2*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, *int widthHeight*, int *count*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectors_ma_64f_L (const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride1*, int *srcStride2*, Ipp32f** *ppDstVectorsRight*, int *dstRightRoiShift*, int *dstRightStride1*, int *dstRightStride2*, Ipp32f** *ppDstVectorsLeft*, int *dstLeftRoiShift*, int *dstLeftStride1*, int *dstLeftStride2*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

Case 5: Eigenvalues and right eigenvectors for a matrix array

IppStatus ippmEigenValuesVectorsRight_ma_32f (const Ipp32f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp32f* *pDstVectorsRight*, int *dstRightStride0*, int *dstRightStride1*, int *dstRightStride2*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectorsRight_ma_64f (const Ipp64f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp64f* *pDstVectorsRight*, int *dstRightStride0*, int *dstRightStride1*, int *dstRightStride2*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectorsRight_ma_32f_P (const Ipp32f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, Ipp32f** *ppDstVectorsRight*, int *dstRightRoiShift*, int *dstRightStride0*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectorsRight_ma_64f_P (const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride0*, Ipp64f** *ppDstVectorsRight*, int *dstRightRoiShift*, int *dstRightStride0*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectorsRight_ma_32f_L (const Ipp32f** ppSrc, int *srcRoiShift*, int *srcStride1*, int *srcStride2*, Ipp32f** *ppDstVectorsRight*, int *dstRightRoiShift*, int *dstRightStride1*, int *dstRightStride2*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValuesVectorsRight_ma_64f_L (const Ipp64f** *ppSrc*, int *srcRoiShift*, int *srcStride1*, int *srcStride2*, Ipp64f** *ppDstVectorsRight*, int *dstRightRoiShift*, int *dstRightStride1*, int *dstRightStride2*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

Case 6: Eigenvalues and left eigenvectors for a matrix array

IppStatus ippmEigenValuesVectorsLeft_ma_32f (const Ipp32f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp32f* *pDstVectorsLeft*, int *dstLeftStride0*, int *dstLeftStride1*, int *dstLeftStride2*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

```
IppStatus ippmEigenValuesVectorsLeft_ma_64f (const Ipp64f* pSrc, int srcStride0, int
srcStride1, int srcStride2, Ipp64f* pDstVectorsLeft, int dstLeftStride0, int dstLeftStride1,
int dstLeftStride2, Ipp64f* pDstValuesRe, Ipp64f* pDstValuesIm, int widthHeight, int count,
Ipp8u* pBuffer);

IppStatus ippmEigenValuesVectorsLeft_ma_32f_P (const Ipp32f** ppSrc, int srcRoiShift, int
srcStride0, Ipp32f** ppDstVectorsLeft, int dstLeftRoiShift, int dstLeftStride0, Ipp32f*
pDstValuesRe, Ipp32f* pDstValuesIm, int widthHeight, int count, Ipp8u* pBuffer);

IppStatus ippmEigenValuesVectorsLeft_ma_64f_P (const Ipp64f** ppSrc, int srcRoiShift, int
srcStride0, Ipp64f** ppDstVectorsLeft, int dstLeftRoiShift, int dstLeftStride0, Ipp64f*
pDstValuesRe, Ipp64f* pDstValuesIm, int widthHeight, int count, Ipp8u* pBuffer);

IppStatus ippmEigenValuesVectorsLeft_ma_32f_L (const Ipp32f** ppSrc, int srcRoiShift, int
srcStride1, int srcStride2, Ipp32f** ppDstVectorsLeft, int dstLeftRoiShift, int
dstLeftStride1, int dstLeftStride2, Ipp32f* pDstValuesRe, Ipp32f* pDstValuesIm, int
widthHeight, int count, Ipp8u* pBuffer);

IppStatus ippmEigenValuesVectorsLeft_ma_64f_L (const Ipp64f** ppSrc, int srcRoiShift, int
srcStride1, int srcStride2, Ipp64f** ppDstVectorsLeft, int dstLeftRoiShift, int
dstLeftStride1, int dstLeftStride2, Ipp64f* pDstValuesRe, Ipp64f* pDstValuesIm, int
widthHeight, int count, Ipp8u* pBuffer);
```

Parameters

pSrc, *ppSrc*	Pointer to the source matrix or array of matrices.
srcStride0	Stride between matrices in the source array.
srcStride1	Stride between rows in the source matrix(ces).
srcStride2	Stride between elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDstVectorsRight, *ppDstVectorsRight*	Pointer to the destination matrix or array of matrices whose columns are right eigenvectors. The size of each matrix must be at least equal to *widthHeight* * *widthHeight*. See eigenvectors storage features in the Description subsection.
dstRightStride0	Stride between matrices in the destination matrix array of right eigenvectors.
dstRightStride1	Stride between rows in the destination matrix of right eigenvectors.
dstRightStride2	Stride between elements in the destination matrix of right eigenvectors.
dstRightRoiShift	ROI shift in the destination matrix of right eigenvectors.
pDstVectorsLeft, *ppDstVectorsLeft*	Pointer to the destination matrix or array of matrices whose columns are left eigenvectors. The size of each matrix must be at least equal to *widthHeight* * *widthHeight*. See eigenvectors storage features in the Description below.

dstLeftStride0	Stride between matrices in the destination matrix array of left eigenvectors.
dstLeftStride1	Stride between rows in the destination matrix of left eigenvectors.
dstLeftStride2	Stride between elements in the destination matrix of left eigenvectors.
dstLeftRoiShift	ROI shift in the destination matrix of left eigenvectors.
pDstValuesRe	Pointer to the destination dense array containing real parts of eigenvalues. The number of elements in the array must be at least equal to *widthHeight* for a matrix or *widthHeight**`count` for an array of matrices. Note that for a complex conjugate pair of eigenvalues, the real part is stored in the array twice.
pDstValuesIm	Pointer to the destination dense array containing imaginary parts of eigenvalues. The number of elements in the array must be at least equal to *widthHeight* for a matrix or *widthHeight*count* for an array of matrices. Note that in a complex conjugate pair of eigenvalues, the positive imaginary part is stored in the array first.
widthHeight	Size of the source square matrix (matrices).
count	The number of matrices in the array.
pBuffer	Pointer to the allocated buffer used for internal computations. You should compute the buffer size using the function EigenValuesVectorsGetBufSize prior to calling `ippmEigenValuesVectors`.

Description

The function `ippmEigenValuesVectors` is declared in the `ippm.h` header file.

Given a real general (nonsymmetric) square matrix *A* of size *widthHeight*widthHeight*, the function finds eigenvalues λ, right and left eigenvectors, such that

- $A*z=\lambda*z$ for the right eigenvectors z,
- $z^H*A=\lambda*z^H$ for the left eigenvectors z,

where z^H is the conjugate transpose of z.

Real nonsymmetric matrices may have complex eigenvalues. If a real nonsymmetric matrix has a complex eigenvalue *a+ib*, then *a-ib* is also an eigenvalue (*i* is the imaginary unit.).

Real parts of eigenvalues are stored densely in the array pointed by *pDstValuesRe*. and the imaginary parts are stored in the same order densely in the array pointed by *pDstValuesIm*. For a complex conjugate pair of eigenvalues, the real part is stored twice and imaginary parts are stored one after another, the positive one being stored first.

Eigenvectors are stored in the same order as the corresponding eigenvalues in the matrix(ces) pointed by *pDstVectorsLeft* (*ppDstVectorsLeft*) or *pDstVectorsRight* (*ppDstVectorsRight*). If the eigenvalue λ(*j*) is real, then the *j*-th column *v*(*j*) of the storage matrix contains the corresponding real eigenvector *z*(*j*). If the eigenvalues $\lambda(j)$ and $\lambda(j+1)$ make up a complex conjugate pair, the respective columns of the storage matrix do not directly contain the eigenvectors. However, the respective eigenvectors, which also make up the complex conjugate pair, can be obtained from the matrix columns $v(j)$ and $v(j+1)$ as follows:

$z(j) = v(j) + i*v(j+1)$ and $z(j+1) = v(j) - i*v(j+1)$, where i is the imaginary unit.

The number of eigenvectors may be less than the matrix order and is equal to the number of different eigenvalues.

The eigenvectors are normalized using the Euclidean norm:

$$\|z\|_E^2 = \|z\|_2^2 = \sum_i |z_i|^2$$

In case of complex eigenvectors, the complex rotation is also applied to make the largest component real.

When all eigenvalues and eigenvectors are computed, the classical spectral factorization of A is

- $A = R\Lambda R^{-1}$, where R is a matrix whose columns are the right eigenvectors,
- $A = (L^H)^{-1}\Lambda L^H$, where L is a matrix whose columns are the left eigenvectors,

and L is a diagonal matrix whose elements are the eigenvalues.

If the number of different eigenvalues is equal to the matrix order, then the matrix A has linearly independent eigenvectors. In this case, $L^H = R^{-1}$, so the spectral factorization of A is $A = R\Lambda L^H$.

To solve a nonsymmetric eigenvalue problem, first a matrix is reduced to the upper Hessenberg form. Then the eigenvalues and eigenvectors are computed with the Hessenberg matrix obtained using the QR algorithm.

If the QR algorithm has not converged in a given number of iterations, the function returns status `ippStsCountMatrixErr` (see Return Values).

However the calculations continue for the remaining 3D vectors in the source array.

Return Values

`ippStsOk`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one input pointer is NULL.
`ippStsSizeErr`	Indicates an error when the input size parameter is less or equal to 0.
`ippStsStrideMatrixErr`	Indicates an error when any of the stride values is not positive or not divisible by the size of the data type.
`ippStsRoiShiftMatrixErr`	Indicates an error when the RoiShift value is negative or not divisible by the size of the data type.
`ippStsCountMatrixErr`	Indicates an error when the *count* value is less or equal to 0.
`ippStsSingularErr`	Indicates an error when any of the input matrices is singular.
`ippStsConvergeErr`	Indicates an error when the algorithm does not converge.

Example

The code example below demonstrates how to use the function `ippmEigenValuesVectors_m_32f`.

```
IppStatus EigenValuesVectors_m_32f (void) {
    /* Source data: matrix with width=4 and height=4 */
    Ipp32f pSrc[4*4]= {1, 1, 1, 3,
                       2, 1, 3, 1,
                       3, 2, 0, 1,
                       1, 3, 1, 3};

    int widthHeight=4;

    int srcStride1 = 4*sizeof(Ipp32f);
    int srcStride2 = sizeof(Ipp32f);

    Ipp32f pDstVectorsRight[4*4]; /* Right Eigenvectors location */
    Ipp32f pDstVectorsLeft[4*4];  /* Left  Eigenvectors location */

    int dstRightStride1 = 4*sizeof(Ipp32f);
    int dstRightStride2 = sizeof(Ipp32f);
    int dstLeftStride1  = 4*sizeof(Ipp32f);
    int dstLeftStride2  = sizeof(Ipp32f);

    Ipp32f pDstValuesRe[4]; /* Real parts Eigen values location      */

    Ipp32f pDstValuesIm[4]; /* Imaginary parts Eigen values location */



    Ipp8u* pBuffer; /* Pointer to the buffer */
    int SizeBytes;  /* Size of the buffer should be specified */

    IppStatus status;

    /* It is required to get the buffer size */
    status=ippmEigenValuesVectorsGetBufSize_32f(widthHeight, &SizeBytes);
```

```
    /* It is required to allocate the buffer of SizeBytes size */
    pBuffer=ippsMalloc_8u(SizeBytes);

    /* Call EigenValuesVectors function */
    status=ippmEigenValuesVectors_m_32f((const Ipp32f*)pSrc,
        srcStride1, srcStride2, pDstVectorsRight, dstRightStride1,
        dstRightStride2, pDstVectorsLeft, dstLeftStride1, dstLeftStride2,
        pDstValuesRe, pDstValuesIm, widthHeight, pBuffer);

    ippsFree(pBuffer);

    /*
    // It is required for EigenValuesVectors function to check return status

    // for catching wrong result in case of invalid input data
    */
    if (status == ippStsOk) {

        printf_m_Ipp32f("Right Eigenvectors matrix:", pDstVectorsRight, 4, 4,
             status);
        printf_m_Ipp32f("Left Eigenvectors matrix:", pDstVectorsLeft, 4, 4,
             status);

        printf_va_Ipp32f("Eigenvalues real parts:", pDstValuesRe, 4, 1,
            status);
        printf_va_Ipp32f("Eigenvalues imaginary parts:", pDstValuesIm,
            4, 1, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }

    return status;
}
```

Output:

```
Right Eigenvectors matrix:
0.474274  0.634963  -0.000000  0.427198
0.517958  -0.265734  0.248532  0.490144
0.382597  0.137734  0.336858  -0.734311
0.600337  -0.374312  -0.438048  -0.195057
Left Eigenvectors matrix:
0.464708  -0.418516  -0.286199  0.261720
0.481775  0.588117  -0.000000  0.578442
0.431235  0.235992  -0.272731  -0.735273
0.604960  -0.327181  0.399914  -0.237237
Eigenvalues real parts:
6.870119  0.224240  0.224240  -2.318601
Eigenvalues imaginary parts:
0.000000  1.684493  -1.684493  0.000000
```

EigenValues

Finds eigenvalues for real general (nonsymmetric) matrices.

Syntax

Case 1: Matrix operation

IppStatus ippmEigenValues_m_32f (const Ipp32f* *pSrc*, int *srcStride1*, int *srcStride2*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValues_m_64f (const Ipp64f* *pSrc*, int *srcStride1*, int *srcStride2*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValues_m_32f_P (const Ipp32f** *ppSrc*, int *srcRoiShift*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValues_m_64f_P (const Ipp64f** *ppSrc*, int *srcRoiShift*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, Ipp8u* *pBuffer*);

Case 2: Matrix array operation

IppStatus ippmEigenValues_ma_32f (const Ipp32f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValues_ma_64f (const Ipp64f* *pSrc*, int *srcStride0*, int *srcStride1*, int *srcStride2*, Ipp64f* *pDstValuesRe*, Ipp64f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

IppStatus ippmEigenValues_ma_32f_P (const Ipp32f** ppSrc, int *srcRoiShift*, int *srcStride0*, Ipp32f* *pDstValuesRe*, Ipp32f* *pDstValuesIm*, int *widthHeight*, int *count*, Ipp8u* *pBuffer*);

```
IppStatus ippmEigenValues_ma_64f_P (const Ipp64f** ppSrc, int srcRoiShift, int srcStride0,
Ipp64f* pDstValuesRe, Ipp64f* pDstValuesIm, int widthHeight, int count, Ipp8u* pBuffer);

IppStatus ippmEigenValues_ma_32f_L (const Ipp32f** ppSrc, int srcRoiShift, int srcStride1,
int srcStride2, Ipp32f* pDstValuesRe, Ipp32f* pDstValuesIm, int widthHeight, int count,
Ipp8u* pBuffer);

IppStatus ippmEigenValues_ma_64f_L (const Ipp64f** ppSrc, int srcRoiShift, int srcStride1,
int srcStride2, Ipp64f* pDstValuesRe, Ipp64f* pDstValuesIm, int widthHeight, int count,
Ipp8u* pBuffer);
```

Parameters

pSrc, *ppSrc*	Pointer to the source matrix or array of matrices.
srcStride0	Stride between matrices in the source array.
srcStride1	Stride between rows in the source matrix(ces).
srcStride2	Stride between elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDstValuesRe	Pointer to the dense destination array containing real parts of eigenvalues. The number of elements in the array must be at least equal to *widthHeight* for a matrix or *widthHeight*count* for an array of matrices. Note that for a complex conjugate pair of eigenvalues, the real part is stored in the array twice.
pDstValuesIm	Pointer to the dense destination array containing imaginary parts of eigenvalues. The number of elements in the array must be at least equal to *widthHeight* for a matrix or *widthHeight*count* for an array of matrices. Note that in a complex conjugate pair of eigenvalues, the positive imaginary part is stored in the array first.
widthHeight	Size of the source square matrix (matrices).
count	The number of matrices in the array.
pBuffer	Pointer to the allocated buffer used for internal computations. You should compute the buffer size using the function EigenValuesGetBufSize prior to calling *ippmEigenValues*.

Description

The function `ippmEigenValues` is declared in the `ippm.h` header file.

Given a real general (nonsymmetric) square matrix *A* of size `widthHeight*widthHeight`, the function finds eigenvalues λ such that

- $A*z=\lambda*z$ for the right eigenvectors *z*,
- $z^H*A=\lambda*z^H$ for the left eigenvectors z,

where z^H is the conjugate transpose of z.

Real parts of eigenvalues are stored densely in the array pointed by *pDstValuesRe* and the imaginary parts are stored in the same order densely in the array pointed by *pDstValuesIm*. For a complex conjugate pair of eigenvalues, the real part is stored twice and imaginary parts are stored one after another, the positive one being stored first.

Return Values

`ippStsOk`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one input pointer is NULL.
`ippStsSizeErr`	Indicates an error when the input size parameter is less or equal to 0.
`ippStsStrideMatrixErr`	Indicates an error when any of the stride values is not positive or not divisible by the size of the data type.
`ippStsRoiShiftMatrixErr`	Indicates an error when the RoiShift value is negative or not divisible by the size of the data type.
`ippStsCountMatrixErr`	Indicates an error when the *count* value is less or equal to 0.
`ippStsSingularErr`	Indicates an error when any of the input matrices is singular.
`ippStsConvergeErr`	Indicates an error when the algorithm does not converge.

Example

The code example below demonstrates how to use the function `ippmEigenValues_m_32f`.

```
IppStatus EigenValues_m_32f (void) {
    /* Source data: matrix with width=4 and height=4 */
    Ipp32f pSrc[4*4]= {1, 1, 1, 3,
                       2, 1, 3, 1,
                       3, 2, 0, 1,
                       1, 3, 1, 3};

    int widthHeight=4;

    int srcStride1 = 4*sizeof(Ipp32f);
    int srcStride2 = sizeof(Ipp32f);

    Ipp32f pDstValuesRe[4]; /* Real parts of Eigenvalues location   */
    Ipp32f pDstValuesIm[4]; /* Imaginary parts of Eigenvalues location  */

    Ipp8u* pBuffer; /* Pointer to the buffer */
    int SizeBytes;  /* Size of the buffer should be specified */

    IppStatus status;

    /* It is required to get the buffer size */
    status=ippmEigenValuesGetBufSize_32f(widthHeight, &SizeBytes);

    /* It is required to allocate the buffer of SizeBytes size */
    pBuffer=ippsMalloc_8u(SizeBytes);

    /* Call EigenValues function */
    status=ippmEigenValues_m_32f((const Ipp32f*)pSrc,
        srcStride1, srcStride2, pDstValuesRe, pDstValuesIm,
        widthHeight, pBuffer);

    ippsFree(pBuffer);
```

```
    /*
    // It is required for EigenValues function to check return status
    // for catching wrong result in case of invalid input data
    */
    if (status == ippStsOk) {
        printf_va_Ipp32f("Eigenvalues real parts:", pDstValuesRe, 4, 1, status);
        printf_va_Ipp32f("Eigenvalues imaginary parts:", pDstValuesIm,
            4, 1, status);
    } else {
        printf("Function returns status: %s \n", ippGetStatusString(status));
    }
    return status;
}
```

Output:

```
Eigenvalues real parts:
6.870119  0.224240  0.224240  -2.318601
Eigenvalues imaginary parts:
0.000000  1.684493  -1.684493  0.000000
```

EigenValuesVectorsGetBufSize

Computes the work buffer size for the functions EigenValuesVectors.

Syntax

IppStatus ippmEigenValuesVectorsGetBufSize_32f (int *widthHeigh*, int* *pSizeBytes*);

IppStatus ippmEigenValuesVectorsGetBufSize_64f (int *widthHeight*, int* *pSizeBytes*);

Parameters

widthHeight	Size of the square matrix.
pSizeBytes	Pointer to the work buffer size value in bytes.

Description

The function `ippmEigenValuesVectorsGetBufSize` is declared in the `ippm.h` file. This function computes the size in bytes of the work buffer that is required for the functions EigenValuesVectors and stores the result at the address of *pSizeBytes*. To compute the size of the buffer, you must specify the size (*widthHeight*) of the square matrix that you are going to compute eigenvalues and eigenvectors for.

Return Values

`ippStsOk`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the input pointer is NULL.
`ippStsSizeErr`	Indicates an error when the input size parameter is less or equal to 0.

EigenValuesGetBufSize

Computes the work buffer size for the functions EigenValues.

Syntax

```
IppStatus ippmEigenValuesGetBufSize_32f (int widthHeight, int* pSizeBytes);
IppStatus ippmEigenValuesGetBufSize_64f (int widthHeight, int* pSizeBytes);
```

Parameters

widthHeight	Size of the square matrix.
pSizeBytes	Pointer to the work buffer size value in bytes.

Description

The function `ippmEigenValuesGetBufSize` is declared in the `ippm.h` file.

This function computes the size in bytes of the work buffer that is required for the functions EigenValues and stores the result at the address of *pSizeBytes*. To compute the size of the buffer, you must specify the size (*widthHeight*) of the square matrix that you are going to compute eigenvalues for.

Return Values

`ippStsOk`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the input pointer is NULL.
`ippStsSizeErr`	Indicates an error when the input size parameter is less or equal to 0.

Realistic Rendering and 3D Data Processing

9

This chapter describes the Intel® Integrated Performance Primitives (Intel® IPP) for realistic rendering and 3D data processing.

The table below lists functions described in more detail later in this chapter:

Functions for Realistic Rendering

Function Base Name	Operation
Ray-Scene Intersection Engine	
IntersectMO	Calculates parameters of intersection of rays with the scene triangles.
IntersectEyeSO	Calculates intersection of the primary ray with the geometry of scene.
IntersectAnySO	Performs occlusion tests for block of rays with the single origin.
IntersectMultipleSO	Calculates the parameters of intersection of rays with the specified number of scene triangles.
Ray-Casting Functions	
CastEye	Calculates the vectors of direction for primary rays.
CastReflectionRay	Calculates the vectors of direction for secondary rays.
CastShadowSO	Calculates the vectors of direction for shadow rays.
Surface Properties Functions	
SurfFlatNormal	Calculates the flat surface normals.
SurfSmoothNormal	Calculates the smooth surface normals.
HitPoint3DEpsSO	Calculates coordinates of the hit points for a block of rays from the single origin.
HitPoint3DEpsMO	Calculates coordinates of the hit points for a block of rays from the multiple origins.
Shader Support Functions	
Dot	Calculates the dot product of two vectors.
DotChangeNorm	Calculates the dot product of two vectors and changes the sign of the surface normal.
Mul	Multiplies accumulator and source vectors.
AddMulMul	Multiplies two source vectors and adds product to the accumulator.
Divi	Divides two vectors.
DistAttenuationSO	Calculates the distance between source point and intersection points.
Acceleration Functions	
TriangleAccelInit	Initializes the structure `IpprTriangleAccel`.
TriangleAccelGetSize	Calculates the size of the external buffer for the structure `IpprTriangleAccel`.
SetBoundBox	Calculates the coordinates of the axis aligned bounding box.
KDTreeBuildAlloc	Builds k-D tree for triangles.
KDTreeFree	Frees memory allocated for k-D tree.
Auxiliary Functions	

Function Base Name	Operation
TriangleNormal	Calculates the triangle normals.
Spherical Harmonic Transform Functions	
SHGetSize	Calculates the size of the state structure for spherical harmonic transforms.
SHInit	Initializes the state structure for spherical harmonic transforms.
SH, SHBand	Computes the spherical harmonic functions.
SHTFwd	Computes the forward spherical harmonic transform.
SHTInv	Computes the inverse spherical harmonic transform.
3D Transforms Functions	
ResizeGetBufSize	Calculates the size of the external work buffer.
Resize	Resizes the source volume.
WarpAffineGetBufSize	Calculates the size of the external buffer for the affine transform.
WarpAffine	Performs the general affine transform of the source volume.
Remap	Performs the look-up coordinate mapping of the elements of the source volume.
3D General Linear Filters	
FilterGetBufSize	Calculates the size of the working buffer.
Filter	Filters a volume using a general cuboidal kernel.

Intel IPP Realistic Rendering Objects

This section contains the descriptions of the objects that are used in the Intel IPP realistic rendering functions.

Scene

In the current implementation the scene is presented as a set of the *triangles* and per vertexes *normals* (if they are required for the description of the scene).

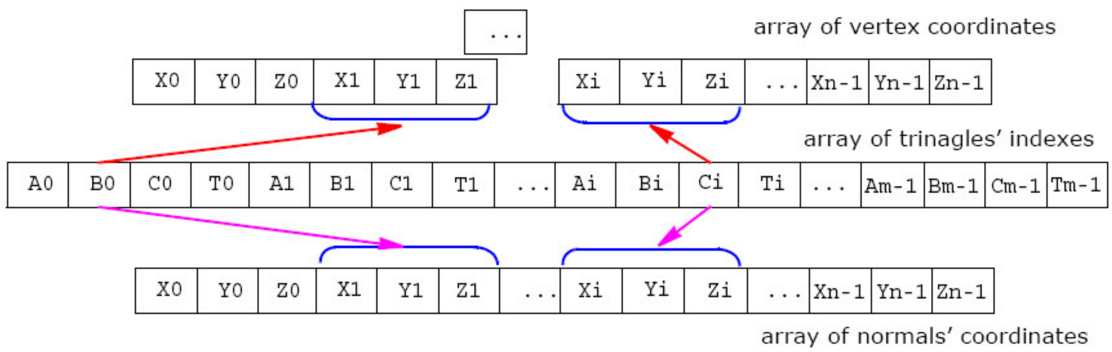
Triangles are specified by two arrays:

- array of vertexes coordinates;
- array of indexes of vertexes.

Each vertex of the triangle is specified by its coordinates *X*, *Y*, and *Z* in the 3D space (Euclidean space). These coordinates are stored in the array of vertexes coordinates (see Figure "Structure of Arrays for Triangle Description").

The description of the triangles consists of a four indexes: A, B, C, T. A, B, C - indexes of vertexes in the array of triangle vertexes coordinates, and T - index in the array of textures (reserved, is not used now).

Structure of Arrays for Triangle Description



These indexes are stored in the array of triangles indexes (see Figure "Structure of Arrays for Triangle Description"). For example, Figure "Structure of Arrays for Triangle Description" shows that vertex B of the triangle 0 has coordinates *X1*, *Y1*, and *Z1* from the array of vertex coordinates.

Normals are specified by two arrays:

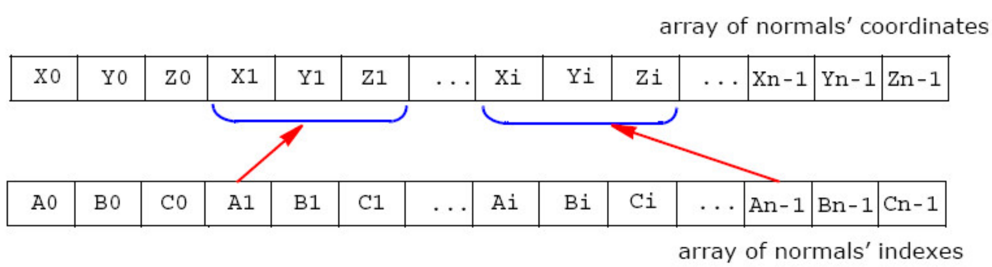
- array of coordinates of the normals;
- array of indexes of normals.

Each normal to the vertex of the triangle is specified by its coordinates *X*, *Y*, and *Z* in the 3D space (Euclidean space). These coordinates are stored in the array of normals coordinates (see Figure "Structure of Arrays for Normal Description").

The description of the normals consists of a three indexes: A, B, C, that are indexes of vertexes in the array of triangle vertexes coordinates. These indexes are stored in the array of normals indexes (see Figure "Structure of Arrays for Normal Description").

Alternatively the normals coordinates can be specified using the array of triangles indexes. Note that in this case the array of normals coordinates should correspond to the array of vertexes coordinates (see Figure "Structure of Arrays for Triangle Description").

Structure of Arrays for Normal Description



Structures and Enumerators

This section describes the structures and enumerators used by the Intel Integrated Performance Primitives for realistic rendering.

The enumerator `IpprIndexType` is used by the function `ipprSurfSmoothNormal` and specifies the index type that indicates how to access to the array of normals.

```
typedef enum {

    ippNormInd = 3,
    ippTriInd  = 4;
} IpprIndexType;
```

The structure `IpprIntersectContext` for representing an intersection context that is used by the intersection engine functions is defined as

```
typedef struct IntersectContext{
    IppBox3D_32f        *pBound;
    IpprTriangleAccel   *pAccel;
    IpprKDTreeNode      *pRootNode;
}IpprIntersectContext;
```

where

pBound is a pointer to the axis aligned bounding box of current object; `pAccel` is a pointer to triangle acceleration structure; and `pRootNode` is a pointer to the root node of the KD-tree.

The following structures are used by the function `ipprKDTreeBuildAlloc`.

The structure `IpprKDTreeNode` for representing a KD-tree node is defined as

```
typedef struct KDTreeNode{
    Ipp32s  flag_k_ofs;
    union _tree_data{
        Ipp32f  split;
        Ipp32s  items;
    }tree_data;
}IpprKDTreeNode;
```

The structure `IpprKDTreeBuildAlg` for tree building algorithm identifiers is defined as

```
typedef enum {
    ippKDTBuildSimple    = 0x499d3dc2,  // Simple building mode
    ippKDTBuildPureSAH   = 0x2d07705b   // SAH building mode
}IpprKDTreeBuildAlg;
```

The structure `IpprSmplBldContext` for setting a simple building mode is defined as

```
typedef struct SimpleBuilderContext{
    IpprKDTreeBuildAlg   Alg;
    Ipp32s               MaxDepth;
}IpprSmplBldContext;
```

where

`Alg` must be equal to `ippKDTBuildSimple` constant;

`MaxDepth` is reserved.

The structure `IpprPSAHBuilderContext` for setting a simple building mode is defined as

```
typedef struct PSAHBuilderContext{
    IpprKDTreeBuildAlg   Alg;
    Ipp32s               MaxDepth;
    Ipp32f               QoS;
    Ipp32s               AvailMemory;
    IppBox3D_32f        *Bounds;
}IpprPSAHBldContext;
```

where

`Alg` must be equal to `ippKDTBuildPureSAH` constant;

`MaxDepth` is a maximum tree subdivision depth (minimum - 0, maximum - 50);

QoS is a termination criteria modifier (minimum - 0.0, maximum - 1.0);

`AvailMemory` is a maximum available memory in Mb;

`*Bounds` is a cut-off bounding box.

Advanced KD-tree building algorithm (ippKDTBuildPureSAH) is based on recursive subspace subdivision according to *surface area heuristic* (SAH). Recursive subdivision of leaf nodes is made according to the criterion: *best SAH cost* after split is less than *original SAH cost* of the leaf plus *cost of split*. An initial leaf node contains triangles having non-empty intersection with the volume contained in bounding box referenced by the `Bounds` field. The *cost of split* depends on QoS field, the higher QoS, the lower the *cost of split*. In general KD-trees with higher QoS tend to be deeper but more efficient. Additional constraint of the maximal KD-tree depth is defined by the `MaxDepth` field.

During KD-tree construction, additional memory is allocated on demand for internal structures in chunks up to the limit defined by the `AvailMemory` field. Actual allocation might be significantly lower than `AvailMemory`. If demand exceeds this limit, `ippStsMemAllocErr` error is returned. Setting `AvailMemory` size incorrectly may result in crash because no additional checks for memory allocation errors are performed during actual allocation. All internally used memory is freed at the end of the build after KD-tree is packed to the compact format. Since it's impossible to accurately estimate the size of the resulting KD-tree, the best practice is to specify this parameter to the half of the total available memory for a single KD-tree build. In this case you make sure that the resulting tree fits in available memory together with all data needed for construction. For simultaneous KD-trees building in multiple threads you need to adjust the `AvailMemory` limit to match your hardware. If you use larger number of threads to process complex models, you must to reduce this constant.

The following structures are used by the functions for 3D data processing.

The structure `IpprVolume` for storing the size of a cuboid is defined as

```
typedef struct {
    int width;
    int height;
    int depth;
} IpprVolume;
```

where `width` , `height`, and `depth` denote the dimensions of the rectangle in the x-, y- and z directions, respectively.

The structure **IpprCuboid** for storing the geometric position and size of a cuboid is defined as

```
typedef struct {
    int x;
    int y;
    int z;
    int width;
    int height;
```

```
    int depth;
} IpprCuboid;
```

where x, y, z denote the coordinates of the top left corner of the cuboid that has dimensions `width` in the x-direction, `height` in the y-direction, and `depth` in the z-direction.

The structure `IpprPoint` for storing the geometric position of a point is defined as

```
typedef struct {
    int x;
    int y;
    int z;
} IpprPoint;
```

where x, y, and z denote the coordinates of the point.

The structure `IpprSHType` for selecting the algorithm used in the spherical harmonic transforms is defined as

```
typedef enum_IpprSHType {
    ipprSHNormDirect=0,
    ipprSHNormRecurr
} IpprSHType;
```

where `ipprSHNormDirect` specifies *direct* algorithm to compute normalized spherical harmonics, `ipprSHNormRecurr` specifies *recurrent* algorithm to compute normalized spherical harmonics.

Ray-Scene Intersection Engine

IntersectMO

Calculates parameters of intersection of rays with the scene triangles.

Syntax

IppStatus ipprIntersectMO_32f(const Ipp32f* const *pOrigin*[3], const Ipp32f* const *pDirection*[3], Ipp32f* *pDistance*, Ipp32f* *pHit*[2], Ipp32s* *pTrngl*, const IpprIntersectContext* *pContext*, IppiSize *blockSize*);

Parameters

pOrigin	Array of pointers to the coordinates of origin point of rays (input).
pDirection	Array of pointers to the vectors of directions (input).
pDistance	Pointer to the array of distance between the hit point and origin of the rays (input).
pHit	Array of pointers to the local surface parameters (*u*, *v*) at the hit point if the intersection is found (output).
pTrngl	Pointer to the triangle index if the intersection is found. If not it is set to -1 (input/output).
pContext	Pointer to the intersection context.

blockSize	Total number of the rays.

Description

The function `ipprIntersectMO` is declared in the `ippr.h` file. This function calculates the parameters of the intersection between rays and scene triangles. Only rays for which value *pTrngl*`[i][j]` is greater than -1 are considered. Rays are specified by coordinates of their origin *pOrigin* and vectors of their directions *pDirection*. The parameter *blockSize* specifies the number of rays. The parameters of the intersection are the distance *pDistance* from the rays origin to the intersection point with the scene triangle, barycentric coordinates *pHit* of the intersection point, index *pTrngl* of the triangle that is closest to the ray's origin. For each ray the function calculates the intersections only with first triangle that are positioned at the distance not greater than initial value of *pDistance*.

To calculate the explicit coordinates of the intersection points, use the function `ipprHitPoint3DEpsMO`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

IntersectEyeSO

Calculates intersection of the primary ray with the geometry of scene.

Syntax

```
IppStatus ipprIntersectEyeSO_32f(IppPoint3D_32f originEye, const Ipp32f* const
pDirection[3], Ipp32f* pDistance, Ipp32f* pHit[2], int* pTrngl, const IpprIntersectContext*
pContext , IppiSize blockSize);
```

Parameters

originEye	Coordinate of the origin point of rays (input).
pDirection	Array of pointers to the vectors of directions (input).
pDistance	Pointer to the array of distance between the hit point and origin of the rays (input).
pHit	Array of pointers to the local surface parameters (*u*, *v*) at the hit point if the intersection is found (output).
pTrngl	Pointer to the triangle index if the intersection is found. If not it is set to -1 (input/output).
pContext	Pointer to the intersection context.
blockSize	Total number of the rays.

Description

The function `ipprIntersectEyeSO` is declared in the `ippr.h` file. This function calculates the parameters of the intersection of the primary ray with the geometry of scene. Only rays for which value *pTrngl*`[i][j]` is greater than -1 are considered. The parameters are the distance between the origin point and point of intersection with the triangle of scene, barycentric coordinates of the intersection point, and index of the closest to the origin triangle. The function calculates the intersections only with first triangles that are positioned at the distance not greater than initial value of *pDistance*.

The code example demonstrates how to use this function.

To calculate the explicit coordinates of the intersection points, use the function `ripprHitPoint3DEpsS0`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

IntersectAnySO

Performs occlusion tests for block of rays with the single origin.

Syntax

```
IppStatus ipprIntersectAnySO_32f(IppPoint3D_32f originEye, const Ipp32f* const pDirection[3]
Ipp32s* pOccluder, Ipp32s* pMask, IppiSize blockSize, const IpprIntersectContext* pContext);
```

Parameters

originEye	Coordinate of the origin point of rays. All rays have the same origin (input).
pDirection	Array of pointers to the vectors of directions, they should not be normalized (input).
pOccluder	Pointer to the array of occluders (otput).
pMask	Pointer to the array of masks (input/output).
pContext	Pointer to the intersection context.
blockSize	Total number of the rays.

Description

The function `ipprIntersectAnySO` is declared in the `ippr.h` file. This function performs occlusion tests - it checks if the scene triangle lays between the ray's origin and the ray's projection on the surface. Only rays for which value *pMask*`[i][j]` is greater than -1 are considered. Indexes of such triangles - occluders - are stored in the array *pOccluder*. If such triangle does not exist for a given ray, its index is set to -1, and the corresponding element of *pMask* is set to -1. It means that the ray is not included in the further consideration.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

IntersectMultipleSO

Calculates the parameters of intersection of rays with the specified number of scene triangles.

Syntax

```
IppStatus ipprIntersectMultipleSO_32f(IppPoint3D_32f originEye, const Ipp32f* const
pDirection[3], Ipp32f* pDistance, Ipp32f* pHit[2], Ipp32s* pTrngl, IpprVolume blockVolume,
const IpprIntersectContext* pContext);
```

Parameters

originEye	Coordinate of the origin point of rays (input).
pDirection	2D array of pointers to the vectors of directions (input).
pDistance	Pointer to the 3D array of distances between the hit point and the origin of the rays (input).
pHit	3D array of pointers to the local surface parameters (*u*, *v*) at the hit point if the intersection is found (output).
pTrngl	(input/output) Pointer to the 3D array of triangle indexes if the intersection is found. If not - it is set to -1 (input/output).
blockVolume	*blockVolume.width * blockVolume.height* is a total number of the rays, *blockVolume.depth* - is the specified number of the scene triangles.
pContext	Pointer to the intersection context.

Description

The function `ipprIntersectMultipleSO` is declared in the `ippr.h` file. This function computes the parameters of the intersection between rays and scene triangles. Only rays for which value *pTrngl*[i][j] is greater than -1 are considered. Rays are specified by coordinates of their origins *originEye* and vectors of their directions *pDirection*. The parameter *blockVolume* specifies the number of rays(*width***height*) and the number of the closest triangles to ray origin that is defined by the *depth*. The parameters of the intersection are the distances *pDistance* from the rays origin to the intersection point with the scene triangles, barycentric coordinates *pHit* of the intersection points, and indexes *pTrngl* of the triangles.

For each ray the function computes the intersection only with the first of *depth* triangles that are positioned at the distance not greater than the initial value of *pDistance*.

The number of triangles that are intersected by the ray in the scene can be less than or equal to *depth*. You can find the actual number of triangles intersected by the ray: in the array *pTrngl* only indexes that are greater than or equal to 0 corresponds to the intersected triangles.

To calculate the explicit coordinates of the intersection points, use the function `ipprHitPoint3DEpsS0`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

Ray Casting Functions

CastEye

Calculates the vectors of direction for primary rays.

Syntax

```
IppStatus ipprCastEye_32f(IppPoint3D_32f imPlaneOrg, IppPoint3D_32f dW, IppPoint3D_32f
dH, int wB,int hB, IppiSize cBlock, Ipp32f* pDirection[3], IppiSize blockSize);
```

Parameters

imPlaneOrg	Coordinate of the projection of the origin point to the projection plane.
dW	Step (vector) along width of the projection plane.
dH	Step (vector) along height of the projection plane.
wB	Number of block along the width of the projection plane.
hB	Number of block along the height of the projection plane.
cBlock	Total number of rays in the block.
pDirection	Array of pointers to separate coordinate (x, y, z) planes of the destination vector.
blockSize	Total number of the rays in the current block.

Description

The function `ipprCastEye` is declared in the `ippr.h` file. This function calculates the vector of direction for the ray as the displacement relative to the point *imPlaneOrg* in accordance with the formula:

pDirection[*i*][*j*] = *imPlaneOrg*[*i*][*j*] + (*wB* + *j*)**dW* + (*hB* + *i*)**dH*,

where

i = 0 .. (*blocSize.height - 1*),

j = 0 .. (*blocSize.width - 1*).

The code example demonstrates how to use this function.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if *pDirection* pointer is `NULL`.

CastReflectionRay

Calculates the vectors of direction for secondary rays.

Syntax

```
IppStatus ipprCastReflectionRay_32f(const Ipp32f* const pIncident[3], const Ipp32s* pMask,
const Ipp32f* const pSurfNorm[3], Ipp32f* pDirection[3], int len);
```

Parameters

pIncident	Pointer to the array of pointers to separate coordinate (x, y, z) planes of the incident rays.
pMask	Pointer to the array of masks.
pSurfNorm	Pointer to the array of pointers to separate coordinate (x, y, z) planes of the normals at the intersection point.
pDirection	Pointer to the array of pointers to separate coordinate (x, y, z) planes of the destination vector.
len	Number of rays.

Description

The function `ipprCastReflectionRay` is declared in the `ippr.h` file. This function calculates the vector of direction of the reflected ray in accordance with the formula:

R = *I* - 2 * <*I***N*> * *N*,

where *R* is the reflected vector, *I* is the incident vector, *N* is the normal vector.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

CastShadowSO

Calculates the vectors of direction for shadow rays.

Syntax

```
IppStatus ipprCastShadowSO_32f( IppPoint3D_32f pOrigin, const Ipp32f* pSurfDotIn, const
Ipp32f* const pSurfNorm[3], const Ipp32f* const pSurfHit[3], Ipp32s* pMask, Ipp32f* pDotRay,
Ipp32f* pDirection[3], int len);
```

Parameters

pOrigin	Pointer to coordinate of the origin.
pSurfDotIn	Pointer to the array of dot products of incident rays and normals at the intersections points.
pSurfNorm	Pointer to the array of pointers to separate coordinates (*x, y, z*) planes of normals at intersections point.

pSurfHit	Pointer to the array of pointers to separate coordinates (*x*, *y*, *z*) planes of the intersection points.
pMask	Pointer to the array of masks.
pDotRay	Pointer to the array of dot products of shadow rays and normals.
pDirection	Pointer to the array of pointers to separate coordinates (*x*, *y*, *z*) planes of the destination vector, they should not be normalized.
len	Number of rays.

Description

The function `opprCastShadowSO` is declared in the `ippr.h` file. This function calculates the array of direction vectors *pDirection* of the shadow rays that is the absolute values of dot products of the shadow rays and normals in the points where the *pMask* is greater than or equal to 0. If the dot product of incident ray and normal, and dot product of shadow ray and normal have different signs, the corresponding values of *pMask* are set to -1. Shadow rays are computed by two points: the origin *pOrigin* and the intersection with the surface *pSurfHit*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if *pDirection* pointer is `NULL`.

Surface Properties Functions

SurfFlatNormal

Calculates the flat surface normals.

Syntax

```
IppStatus opprSurfFlatNormal_32f(const Ipp32f* pTrnglNorm, const Ipp32s* pTrngl, Ipp32f*
pSurfNorm[3], int len);
```

Parameters

pTrngNorm	Pointer to the array of the triangles' normasl.
pTrngl	Pointer to the array of the triangles' indexes.
pSurfNorm	Array of pointers to separate coordinate (*x, y, z*) planes of surface normals at intersections points.
len	Number of rays in the block.

Description

The function `opprSurfFlatNormal` is declared in the `ippr.h` file. This function calculates the flat surface normals in the points of intersection of rays with triangles. In fact the function copies pre-computed triangles' normals from the *pTrngNorm* to the array of the surface normals *pSurfNorm*.

The code example demonstrates how to use this function.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

SurfSmoothNormal

Calculates the smooth surface normals.

Syntax

```
IppStatus ipprSurfSmoothNormal_32f(const Ipp32f* pVertNorm, const Ipp32s* pIndexNorm,
const Ipp32s* pTrngl, const Ipp32f* const pHit[2], Ipp32f* pSurfNorm[3], int len,
IpprIndexType ippInd);
```

Parameters

pVertNorm	Pointer to the vertex normals.
pIndexNorm	Pointer to the normals' indexes.
pTrngl	Pointer to the triangles' indexes.
pHit	Pointer to the array of pointers to the local surface parameter (*u*, *v*) planes at the hit point if the intersection is found.
pSurfNorm	Pointer to the array of pointers to separate coordinate (*x*, *y*, *z*) planes of surface normals at intersections points.
len	Number of rays in the block.
ippInd	Specifies the type of indexing; the following values are possible:
	`ippNormInd` using an array of normals indexes;
	`ippTriInd` using an array of the triangles' indexes.

Description

The function *ipprSurfSmoothNormal* is declared in the `ippr.h` file. This function calculates the surface's smooth normals in the points of intersection of rays with triangles. The function used the linear interpolation of per vertex normal in the intersection point according to the following formulas (in vector representation):

If for triangle *ABC*, *aN*, *bN*, *cN* are per vertex normals, and u, v are barycentric coordinates, then

$uN = aN * u + (1 - u) * cN;$

$vN = bN * v + (1 - v) * cN;$

$uvN = aN * u + bN * v + (1 - (u + v)) * cN$

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

HitPoint3DEpsS0

Calculates coordinates of the hit points for a block of rays from the single origin.

Syntax

```
IppStatus ipprHitPoint3DEpsS0_32f_M(const IppPoint3D_32f originEye, const Ipp32f* const
pDirection[3], const Ipp32f* pDistance, const Ipp32s* pMask, Ipp32f* pSurfHit[3], int len,
Ipp32f eps);
```

Parameters

originEye	Coordinate of the origin point of rays. All rays have the same origin.
pDirection	Pointer to the array of pointers to separate coordinates (*x, y, z*) planes of the ray's directions.
pDistance	Pointer to the generalized distance from origin to intersection point.
pMask	Pointer to the array of masks.
pSurfHit	Pointer to the array of pointers to a separate coordinates (*x, y, z*) planes of the intersection points.
len	Number of rays in the block.
eps	Tolerance value.

Description

The function `ipprHitPoint3DEpsS0` is declared in the `ippr.h` file. For an array of rays from the single origin *originEye* this function calculates the explicit coordinates of the intersection points where the *pMask[i]* is greater than or equal to 0. The array of mask is the array of indexes of the triangles that intersect with the ray.

The tolerance value *eps* help to avoid the numerical imprecision in the intersection defining.

For example, for reflected rays value *eps* = 0.999f, and for refracted and transparency rays value *eps* = 1.001f.

```
pSurfHit[i][0] = originEye[0] + pDirection [i][0] * eps * pDistance[i][0]
pSurfHit[i][1] = originEye[0] + pDirection [i][1] * eps * pDistance[i][1]
pSurfHit[i][2] = originEye[0] + pDirection [i][2] * eps * pDistance[i][2]
```

where `i = 0..(len - 1)`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

HitPoint3DEpsMO

Calculates coordinates of the hit points for a block of rays from the multiple origins.

Syntax

```
IppStatus ipprHitPoint3DEpsMO_32f_M(const Ipp32f* const pOrg[3], const Ipp32f* const
pDirection[3], const Ipp32f* pDistance, const Ipp32s* pMask, Ipp32f* pSurfHit[3], int len,
Ipp32f eps);
```

Parameters

pOrg	Pointer to the array of pointers to separate coordinate (*x, y, z*) planes of the origin points.
pDirection	Pointer to the array of pointers to separate coordinate (*x, y, z*) planes of the ray's directions.
pDistance	Pointer to the generalized distance from origin to intersection point.
pMask	Pointer to the array of masks.
pSurfHit	Pointer to the array of pointers to a separate coordinate (*x, y, z*) planes of the intersection points.
len	Number of rays in the block.
eps	Tolerance value.

Description

The function `ipprHitPoint3DEpsMO` is declared in the `ippr.h` file. For an array of rays from the multiple origins *pOrg* this function calculates the explicit coordinates of the intersection points where the *pMask[i]* is greater than or equal to 0. The array of mask is the array of indexes of the triangles that intersect with the ray.

The tolerance value *eps* help to avoid the numerical imprecision in the intersection defining.

For example, for reflected rays value *eps* = 0.999f, and for refracted and transparency rays value *eps* = 1.001f.

```
pSurfHit[i][0] = pOrg[i][0] + pDirection [i][0] * eps * pDistance[i][0]
pSurfHit[i][1] = pOrg[i][1] + pDirection [i][1] * eps * pDistance[i][1]
pSurfHit[i][2] = pOrg[i][2] + pDirection [i][2] * eps * pDistance[i][2]
```

where `i = 0..(len - 1)`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

Shaders Support Functions

Dot

Calculates the dot product of two vectors.

Syntax

```
IppStatusipprDot_32f_P3C1M(const Ipp32f* const pSrc0[3], const Ipp32f* const pSrc1[3],
const Ipp32s* pMask, Ipp32f* pDot, int len);
```

Parameters

pSrc0	Pointer to the array of pointers to separate coordinate (*x, y, z*) planes of the first source points.
pSrc1	Pointer to the array of pointers to separate coordinate (*x, y, z*) planes of the second source points.
pMask	Pointer to the array of masks.
pDot	Pointer to the destination array of dot product values.
len	Number of rays.

Description

The function `ipprDot` is declared in the `ippr.h` file. This function calculates the dot product of two source vectors in the points where the *pMask [i]* is greater than or equal to 0; in other points the dot product is set to `0.f`. The array of mask is the array of indexes of the triangles that intersect with the ray.

The code example demonstrates how to use this function.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

DotChangeNorm

Calculates the dot product of two vectors and changes the sign of the surface normal.

Syntax

```
IppStatus ipprDotChangeNorm_32f_IM(const Ipp32f* const pSrc[3], const Ipp32s* pMask,
Ipp32f* pSrcDst[3], Ipp32f* pDot, int len);
```

Parameters

pSrc	Pointer to the array of pointers to separate coordinate (*x, y, z*) planes of the first source points.

pSrcDst	Pointer to the array of pointers to separate coordinate (*x, y, z*) planes of the second source and destination points.
pMask	Pointer to the array of masks.
pDot	Pointer to the destination array of dot product values.
len	Number of rays.

Description

The function *ipprDotChangeNorm* is declared in the `ippr.h` file. This function calculates the dot product of two source vectors *pSrc* and *pSrcDst* in the points where the *pMask [i]* is greater than or equal to 0; in other points the dot product is set to *0.f*. The array of mask is the array of indexes of the triangles that intersect with the ray. If dot product value *pDot*[*i*]>0.f, the function changes the sign of *pDot*[*i*] and *pSrcDst*[*i*], that is the surface normal is reversed in direction to the origin point of the ray.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

Mul

Multiplies accumulator and source vectors.

Syntax

```
IppStatus ipprMul_32f_C1IM(const Ipp32f* pSrc, const Ipp32s* pMask, Ipp32f* pSrcDst[3],
int len);
```

```
IppStatus ipprMul_32f_C1P3IM(const Ipp32f* pSrc, const Ipp32s* pMask, Ipp32f* pSrcDst[3],
int len);
```

Parameters

pSrc	Pointer to the source array.
pSrcDst	Pointer to the accumulator array for the 1-channel flavor, or pointer to the array of pointers to separate coordinate (*x, y, z*) planes of the accumulator array for planar flavor.
pMask	Pointer to the array of masks.
len	Number of rays.

Description

The function `ipprMul` is declared in the `ippr.h` file. This function multiplies each *i*-th element of the vectors of the accumulator *pSrcDst* by the corresponding element of the source vector *pSrc* in points where the mask *pMask[i]* is greater than or equal to 0. The result is stored in the *pSrcDst*.

The code example demonstrates how to use this function.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is *NULL*.

AddMulMul

Multiplies two source vectors and adds product to the accumulator.

Syntax

```
IppStatus ipprAddMulMul_32f_AC1P3IM(IppPoint3D_32f point, const Ipp32f* pSrc0, const
Ipp32f* const pSrc1[3], const Ipp32s* pMask, Ipp32f* pSrcDst[3], int len);
```

Parameters

point	Source point.
pSrc0	Pointer to the first source array.
pSrc1	Pointer to the array of pointers to separate coordinate (*x, y, z*) planes of the second source points.
pSrcDst	Pointer to the array of pointers to separate coordinate (*x, y, z*) planes of the accumulator array.
pMask	Pointer to the array of masks.
len	Number of of rays.

Description

The function `ipprAddMulMul` is declared in the `ippr.h` file. This function multiplies the elements of source vectors *pSrc0* and *pSrc1*, and stores results in the accumulator *pSrcDst* in accordance with the following formulas:

```
pSrcDst [0][n] += pSrc1[0][ n] *  pSrc0[n] *  point[0],
pSrcDst [1][n] += pSrc1[1][ n] *  pSrc0[n] *  point[1],
pSrcDst [2][n] += pSrc1[2][ n] *  pSrc0[n] *  point[2],
```

where n = `0, 1, 2,.. len -1`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

Divi

Divides two vectors.

Syntax

`IppStatus ipprDiv_32f_C1IM(const Ipp32f* pSrc, const Ipp32s* pMask, Ipp32f* pSrcDst, int len);`

Parameters

pSrc	Pointer to the source vector.
pSrcDst	Pointer to the source and destionation vector.
pMask	Pointer to the array of masks.
len	Number of of rays.

Description

The function *ipprDiv* is declared in the `ippr.h` file. This function divides each *i*-th element of the vector *pSrcDst* by the corresponding element of the source vector *pSrc* in points where the mask *pMask*`[i]` is greater than or equal to 0. The result is stored in the *pSrcDst*.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or a warning.
ippStsNullPtrErr	Indicates an error condition if one of the specified pointers is NULL.

DistAttenuationSO

Calculates the distance between the source point and intersection points.

Syntax

`IppStatus ipprDistAttenuationSO_32f_M(IppPoint3D_32f point, const Ipp32f* const pSurfHit[3], const Ipp32s* pMask, Ipp32f* pDistance, int len);`

Parameters

point	Source point.
pSurfHit	Pointer to the array of pointers to separate coordinate (*x*, *y*,*z*) planes of the intersection points.
pMask	Pointer to the array of masks.
pDistance	Pointer to the computed distances from the source point (origin) to intersection points.
len	Number of of rays.

Description

The function `ipprDistAttenuationSO` is declared in the `ippr.h` file. This function calculates the distances *pDistance* between the source point *point* and those surface points *pSurfHit* where the mask *pMask[i]* is greater than or equal to 0. If the computed distance is less than 0, its value is set to 1.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

Acceleration Functions

TriangleAccelInit

Initializes the structure `IpprTriangleAccel`.

Syntax

```
IppStatus ipprTriangleAccelInit(IpprTriangleAccel* pTrnglAccel, const Ipp32f* pVertexCoord,
const int* pTrnglIndex, int lenTrngl);
```

Parameters

pTrnglAccel	Pointer to the structure `IpprTriangleAccel`.
pVertexCoord	Pointer to the array of vertex coordinates.
pTrnglIndex	Pointer to the triangle's indexes.
lenTrngl	Number of the triangles in the mesh

Description

The function `ipprTriangleAccelInit` is declared in the `ippr.h` file.

This function initialize the structure *pTrnglAccel* in the external buffer. This structure is specifying by the number of triangles *lenTrngl*, their indexes *pTrnglIndex*, and vertex coordinates *pVertexCoord*. The size of the external buffer should be computed by calling the function ipprTriangleAccelGetSize beforehand.

This structure is used by the functions ipprIntersectMO, ipprIntersectEyeSO, and ipprIntersectAnySO to accelerate the searching the intersections of the rays with triangles.

The code example demonstrates how to use this function.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

TriangleAccelGetSize

Calculates the size of the external buffer for the structure `IpprTriangleAccel`.

Syntax

```
IppStatus ipprTriangleAccelGetSize(int* pTrnglAccelSize);
```

Parameters

pTrnglAccelSize	Pointer to the size of the structure `IpprTriangleAccel`.

Description

The function `ipprTriangleAccelInit` is declared in the `ippr.h` file.

This function calculates the size of the external buffer for the structure `IpprTriangleAccel` and store result in the *pTrnglAccelSize*. This function should be called prior to the function ipprTriangleAccelInit.

The code example demonstrates how to use this function.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

SetBoundBox

Calculates the coordinates of the axis aligned bounding box.

Syntax

```
IppStatus ipprSetBoundBox_32f(const Ipp32f* pVertCoor, int lenTri, IppBox3D_32f* pBound);
```

Parameters

pSVertCoor	Pointer to the coordinates of the vertexes of the triangles.
lenTri	Number of triangles in the mesh.
pSBound	Pointer to the coordinate of the axis aligned bounding box.

Description

The function `ipprSetBoundBox` is declared in the `ippr.h` file. This function calculates the coordinates of the axis aligned bounding box.

The code example demonstrates how to use this function.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.

KDTreeBuildAlloc

Builds k-D tree for triangles.

Syntax

```
IppStatus ipprKDTreeBuildAlloc(IpprKDTreeNode** pDstKDTree, const Ipp32f* const pSrcVert,
const Ipp32s* const pSrcTriInx, Ipp32s srcVertSize, Ipp32s srcTriSize, Ipp32s*
pDstKDTreeSize, const void* const pBldContext);
```

Parameters

ppDstKDTree	Pointer to the pointer to the built k-D tree.
pSrcVert	Pointer to the array of the scene element vertexes.
pSrcTriInx	Pointer to the array of indexed scene element triangles.
srcVertSize	Size of the array of vertexes.
srcTriSize	Size of the array of triangles.
pDstKDTreeSize	Pointer to the size of the built tree.
pBldContext	Pointer to the structure that specifies the building algorithm and algorithm-specific parameters.

Description

The function `ipprKDTreeBuildAlloc` is declared in the `ippr.h` file. This function allocates memory and builds the k-D tree for the set of triangles. The function uses one of the predefined construction algorithms controlled by the service parameters via the parameter *pBldContext*. This parameter points to structure of `IpprSmplBldContext` or `IpprPSAHBldContext` type (see Structures and Enumerators for more details).

Passing argument of type `IpprSmplBldContext` with first element `Alg` set to `ippKDTBuildSimple` constant causes simple tree construction algorithm. This algorithm is useful for testing purposes only. It builds single-leafed tree with all triangles associated with this leaf. Passing argument of `IpprPSAHBldContext` with first element `Alg` set to `ippKDTBuildPureSAH` constant causes SAH-based tree construction controlled by other structure parameters. See their detailed description in Structures and Enumerators.

Due to the algorithm specific implementation, initial memory allocation for SAH-based tree building cannot be less than 80Mb even for very small scenes, this limits a minimal useful value of the `AvailMemory` by 81Mb.

It is possible to specify cut-off bounding box that does not enclose all triangles specified by the parameter *pSrcTriInx*. In this case, k-D tree is built only for specified sub-volume, and output tree leaves contain only indexes to triangles intersected by this sub-volume.

The code example demonstrates how to use this function.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if one of the arrays has negative size.
`ippStsOutOfRangeErr`	Indicates an error if `QoS` is out of the range [0.0, 1.0], or `MaxDepth` is out of the range [0, 50].

ippStsNoMemErr	Indicates an error if there is not enough memory during initial allocation.
ippStsNoMemAllocErr	Indicates an error if there is not enough memory for the actual tree building.
ippStsBadArgErr	Indicates an error if the algorithm type is not valid.
ippStsErr	Indicates an internal algorithm error.

KDTreeFree

Frees memory allocated for k-D tree.

Syntax

```
void ipprKDTreeFree(IpprKDTreeNode* pSrcKDTree);
```

Parameters

pSrcKDTree	Pointer to the allocated k-D tree.

Description

The function `ipprKDTreeFree` is declared in the `ippr.h` file. This function frees the memory allocated for the k-D tree by the function ipprKDTreeBuildAlloc.

The code example demonstrates how to use this function.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or a warning.
ippStsNullPtrErr	Indicates an error if the pointer *pSrcKDTree* is NULL.

Auxiliary Functions

TriangleNormal

Calculates the triangle normals.

Syntax

```
IppStatus ipprTriangleNormal_32f(const Ipp32f* pTrnglCoor, const int* pTrnglIndex, Ipp32f*
pTrnglNorm, int lenTrngl);
```

Parameters

pTrnglCoor	Pointer to the coordinates of the vertexes of the triangles.
pTrnglIndex	Pointer to the indexes of the triangles.
pTrnglNorm	Pointer to the normal of triangles.
lenTrngl	Number of triangles in the mesh.

Description

The function `ipprTriangleNormal` is declared in the `ippr.h` file. This function calculates the triangle mesh normals.

The code example demonstrates how to use this function.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.

Spherical Harmonic Transform

This section describes the Intel® IPP function for spherical harmonic transformation.

Here $\{Y_l^m(x, y, z): 0 \le l \le L, |m| \le L\}$ denotes full set of orthogonal spherical harmonic (SH) basic functions up to order L on a unit sphere in Cartesian coordinates. So there exist $(2L+1)$ SH functions for each order l (an SH *band*), and $(L+1)(L+1)$ SH functions for all orders up to L. SH functions are indexed within the order from $-m$ to m.

The forward spherical harmonic transform (SHT) of order L of a function $f(x, y, z)$ on a unit sphere is a set of coefficients $c_l^m: 0 \le l \le L, |m| \le L$, where

$$c_l^m = \oint f(x, y, z) Y_l^m(x, y, z) d\Omega$$

integrating over solid angle.

The inverse SHT

$$\sum_{0 \le l \le L,\ |m| \le L} c_l^m(x, y, z) Y_l^m(x, y, z)$$

of order L is converging to the original function $f(x, y, z)$ when $L \to \infty$.

The SHT of a function $f(x, y, z)$ can be approximated as

$$\tilde{c}_l^m = \sum_{x_i, y_i, z_i} f(x_i, y_i, z_i) Y_l^m(x_i, y_i, z_i)$$

by summation over representative set of a unit sphere points.

API for Spherical Harmonic Functions

The normalized spherical harmonic functions $Y_l^m(x, y, z)$ used in Intel IPP are real-valued polynomials of Cartesian coordinates which can be defined by the following recurrent equations:

$$a_0^0 = 1, \qquad a_1^0 = \sqrt{3}, \qquad a_1^1 = -\sqrt{3}, \qquad a_l^l = \sqrt{\frac{2l+1}{2l}}, \qquad a_l^m = \sqrt{\frac{4l^2+1}{l^2-m^2}},$$

$$b_l^m = -\sqrt{\left(\frac{2l+1}{2l-3} \cdot \frac{(l-1)^2-m^2}{l^2-m^2}\right)},$$

$2 \le l \le L$, $0 \le m < l$.

$$Y_0^0 = \frac{a_0^0}{\sqrt{4\pi}}, \qquad Y_1^{-1} = \frac{a_1^1}{\sqrt{4\pi}} \cdot y\ , \qquad Y_1^0 = \frac{a_1^0}{\sqrt{4\pi}} \cdot z\ , \qquad Y_1^1 = \frac{a_1^1}{\sqrt{4\pi}} \cdot x\ ,$$

$$Y_l^l = a_l^l(-Y_{l-1}^{l-1} \cdot x + Y_{l-1}^{l-1} \cdot y), \qquad Y_l^{-l} = a_l^l(-Y_{l-1}^{-l+1} \cdot y + Y_{l-1}^{-l+1} \cdot x),$$

$$Y_l^{l-1} = (a_l^{l-1} Y_{l-1}^{l-1} \cdot z),$$

$$Y_l^m = a_l^{|m|} Y_{l-1}^m \cdot z + b_l^{|m|} Y_{l-2}^m$$

$2 \le l \le L$, $|m| \le (l-1)$.

These equations can be derived from the three-term recurrence [see M.A.Blanco, M.Florez, M.Bermejo, "*Evaluation of the rotation matrices in the basis of real spherical harmonics*", Journal of Molecular Structure (Theochem), 1997, 419, 19-27, or R.Green, "*Spherical harmonic lighting: The gritty details*", in Game Developers' Conference, 2003] known for the associated Legendre polynomials $(l-m)P^m_l = x(2l-1)P^m_{l-1} - (l-1+m)P^m_{l-2}$ using the normalization factor

$$K_l^m = \sqrt{\frac{2l+1}{4\pi} \cdot \frac{(l-|m|)!}{(l+|m|)!}}$$

Direct formulas for normalized real-valued spherical harmonic polynomials in Cartesian coordinates can be derived from following formula (see D.A. Varshalovich A.N. Moskalev, V.K. Khersonsky, "*Quantum Theory of Angular Momentum*", Singapore: World Scientific Publishing, 1988):

$$r^l Y_l^m(x, y, z) = \sqrt{\frac{2l+1}{4\pi} \cdot (l+m)!(l-m)!} \sum_{p, q, s} \frac{1}{p!q!s!}\left(-\frac{x+iy}{2}\right)^p \left(\frac{x+iy}{2}\right)^q z^s$$

Summation is performed over p, q, s >0 such as $p + q + s = l$, $p - q = m$.

Here $x^2 + y^2 + z^2 = r^2$ are Cartesian coordinates.

SHGetSize

Calculates the size of the state structure for spherical harmonic transforms.

Syntax

```
IppStatusipprSHGetSize_32f(Ipp32u maxL, IppSHType shType, Ipp32u* pSize);
```

Parameters

maxL	Maximum order for spherical harmonic transform supported after initialization of the state structure.
shType	Type of algorithm used for SH calculations, possible values: `ippSHNormDirect` or `ippSHNormRecurr`.
pSize	Pointer to the size of the state structure.

Description

The function `ipprSHGetSize` is declared in the `ippr.h` file. This function calculates the size of the external buffer required for the state structure used in the SHT calculations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error.
`ippStsNullPtrErr`	Indicates an error if *pSize* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error if *maxL* is greater than 15.
`ippStsRangeErr`	Indicates an error condition if *shType* has an illegal value.

SHInit

Initializes the state structure for spherical harmonic transforms.

Syntax

```
IppStatusipprSHInit_32f(IppSHState* pSHState, Ipp32u maxL, IppSHType shType);
```

Parameters

pSHState	Pointer to the external buffer for the SHT state structure.
maxL	Maximum order for spherical harmonic transform supported after initialization of the state structure.
shType	Type of algorithm used for SH calculations, possible values: `ippSHNormDirect` or `ippSHNormRecurr`.

Description

The function `ipprSHInit` is declared in the `ippr.h` file. This function initializes the state structure for spherical harmonic transforms in the external buffer. The size of this buffer cannot be less than the size returned by the function `ipprSHGetSize`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error.
`ippStsNullPtrErr`	Indicates an error if *pSize* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error if *maxL* is greater than 15.
`ippStsRangeErr`	Indicates an error condition if *shType* has an illegal value.

SH, SHBand

Computes the spherical harmonic functions.

Syntax

```
IppStatus ipprSH_32f(const Ipp32f* pX, const Ipp32f* pY, const Ipp32f* pZ, Ipp32u N, const Ipp32f* pDstYlm, Ipp32u L, IppSHState* pSHState);

IppStatus ipprSHBand_32f(const Ipp32f* pX, const Ipp32f* pY, const Ipp32f* pZ, Ipp32u N, const Ipp32f* pDstBandYlm, Ipp32u L);
```

Parameters

pX, *pY*, *pZ*	Pointers to the source vectors representing a unit sphere points in Cartesians coordinates.
N	Number of Cartesians points, that is the length of the source vector.
pDstYlm	Pointer to the destination vector to store SH values computed at given points for all orders up to order *L*, of size *N*(*L*+1)(*L*+1).

pDstBandYlm	Pointer to the destination vector to store SH values computed at given points only for order *L*, of size *N*(2*L*+1).
L	Order, can not be greater than maximum order specified in the function `ипprSHInit`.
pSHState	Pointer to the external buffer for the SHT state structure that must be initialized with maximum order not less than *L*.

Description

The functions `ipprSH` and `ipprSHBand` are declared in the `ippr.h` file. These functions calculates the spherical harmonics functions $\{Y_l^m(x, y, z): 0 \le l \le L, |m| \le L\}$ for each input point (*pX*[i], *pY*[i], *pZ*[i]), $0 \le i < N$, which belong to a unit sphere, that is $pX[i]^2 + pY[i]^2 + pZ[i]^2 = 1$.

`ipprSH`. For each input point this function computes SH function values for all bands up to *L*. The total number of values for each point is (*L*+1)(*L*+1). They are stored in the destination vector *pDstYlm* as follows:

$$pDstYlm\,[i(L+1)(L+1) + 2l + m] = Y_l^m(pX[i], pY[i], pZ[i]), -L \le l \le L,\ |m| \le l,\ 0 \le i < N.$$

That is, they are stored by points, for each point by bands, and for each band by indexes in the following way:

Y_l^m values for the first point, all bands from 0 to *L*

Y_0^0 - 0-band;

Y_1^{-1}, Y_1^0, Y_1^1 - 1-band;

Y_2^{-2}, Y_2^{-1}, Y_2^0 , Y_2^1, Y_2^2 - 2-band

...

Y_L^{-L}, Y_L^{-L+1} ,... Y_L^{-1}, Y_L^0, Y_L^1,... Y_L^{L-1}, Y_L^L - *L*-band.

Then analogous set of Y_l^m values for the second point, all bands from 0 to *L*, then for the third point and so on up to the *N*-th point.

`ipprSHBand`. This function computes SH functions values for each point only for given band *L*. The total number of values for each point is (2*L*+1). They are stored in the destination vector *pDstBandYlm* as follows:

$$pDstBandYlm\,[i(2L+1) + L + m] = y_L^m(pX[i], pY[i], pZ[i]), -L \le m \le L,\ 0 \le i < N.$$

That is, they are stored by points for the given band *L*, and for each point by indexes in the following way:
y_L^{-L}, y_L^{-L+1} ,... y_L^{-1}, y_L^0, y_L^1,... y_L^{L-1}, y_L^L.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *N* is equal to 0.
`ippStsRangeErr`	Indicates an error if *L* is greater that maximum order specified in the function `ipprSHInit`.

SHTFwd

Computes the forward spherical harmonic transform.

Syntax

```
IppStatusripprSHTFwd_32f_C1I(const Ipp32f* pX, const Ipp32f* pY, const Ipp32f* pZ, const
Ipp32f* pSrc, Ipp32u N, Ipp32f* pSrcDstClm, Ipp32u L, IppSHState* pSHState);
```

```
IppStatus ipprSHTFwd_32f_C3I(const Ipp32f* pX, const Ipp32f* pY, const Ipp32f* pZ, const
Ipp32f* pSrc, Ipp32u N, Ipp32f* pSrcDstClm[3], Ipp32u L, IppSHState* pSHState);
```

```
IppStatus ipprSHTFwd_32f_P3I(const Ipp32f* pX, const Ipp32f* pY, const Ipp32f* pZ, const
Ipp32f* pSrc[3], Ipp32u N, Ipp32f* pSrcDstClm[3], Ipp32u L, IppSHState* pSHState);
```

Parameters

pX, *pY*, *pZ*	Pointers to the source vectors representing a unit sphere points in Cartesians coordinates.
pSrc	Pointer to the source vector of values assigned to each point represented by the input vectors.
N	Number of Cartesians points, that is the length of the source vector.
pSrcDstClm	Pointer to the destination vector or an array of pointers to the destination vectors to store the running values of SHT coefficients of length $(L+1)(L+1)$.
L	Order, can not be greater than maximum order specified in the function `ipprSHInit`.
pSHState	Pointer to the external buffer for the SHT state structure that must be initialized with maximum order not less than *L*.

Description

The function `ipprSHTFwd` is declared in the `ippr.h` file.

All function flavors perform projecting of a function $f(x, y, z)$ defined on a unit sphere for each input point (pX[i], pY[i], pZ[i]) into the SH functions basis $Y_l^m(x, y, z)$, $0 \le l \le L$, $|m| \le L$, that is computation of the SHT coefficients $\{C_l^m: 0 \le l \le L, |m| \le L\}$ by accumulating partial sums of SHT integral in the destination vectors.

The function `ipprSHTFwd_32f_C1I` calculates

$$pSrcDstClm[2l+m] += \sum_{li=0}^{N-1} pSrc[i] \cdot Y_l^m(pX[i], pY[i], pZ[i])$$

for $0 \le l \le L$, $|m| \le L$.

The function `ipprSHTFwd_32f_C3P3I` calculates

$$pSrcDstClm3[k][2l+m]+ = \sum_{li=0}^{N-1} pSrc[3i+k] \cdot Y_l^m(pX[i],\ pY[i],\ pZ[i])$$

for $0 \le l \le L$, $|m| \le L$, $0 \le k \le 3$.

The function `ipprSHTFwd_32f_P3I` calculates

$$pSrcDstClm3[k][2l+m]+ = \sum_{li=0}^{N-1} pSrc[k][i] \cdot Y_l^m(pX[i],\ pY[i],\ pZ[i])$$

for $0 \le l \le L$, $|m| \le L$, $0 \le k \le 3$.

The functions `ipprSHTFwd_32f_C3P3I` and `ipprSHTFwd_32f_P3I` are suitable to transform a color function, for example *f:(x, y, z) --> R, G, B*, for both pixel-ordered or planar images respectively.

It is supposed for each input point $(pX[i],\ pY[i],\ pZ[i]),\ 0 \le i < N$, that $pX[i]^2 + pY[i]^2 + pZ[i]^2 = 1$.

The function updates running SHT values c_l^m that are accumulated in the destination vector *pSrcDstClm* or each vector of the destination array *pSrcDstClm* in the following order:

c_0^0

c_1^{-1}, c_1^0, c_1^1

$c_2^{-2}, c_2^{-1}, c_2^0, c_2^1, c_2^2$

...

$c_L^{-L}, c_L^{-L+1}, \ldots c_L^{-1}, c_L^0, c_L^1, \ldots c_L^{L-1}, c_L^L$.

Before the first call to the function `ipprSHTFwd` the destination vector (vectors) *pSrcDstClm* must be zeroed.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *N* is equal to 0.
`ippStsRangeErr`	Indicates an error condition if *L* is greater that maximum order specified in the function `ipprSHInit`.

SHTInv

Computes the inverse spherical harmonic transform.

Syntax

```
IppStatus ipprSHTInv_32f_C1(const Ipp32f* pSrcClm, Ipp32u L, const Ipp32f* pX, const
Ipp32f* pY, const Ipp32f* pZ, Ipp32f* pDst, Ipp32u N, IppSHState* pSHState);
```

```
IppStatus ipprSHTInv_32f_P3C3(const Ipp32f* pSrcClm[3], Ipp32u L, const Ipp32f* pX, const
Ipp32f* pY, const Ipp32f* pZ, Ipp32f* pDst, Ipp32u N, IppSHState* pSHState);
```

```
IppStatus ipprSHTInv_32f_P3(const Ipp32f* pSrcClm[3], Ipp32u L, const Ipp32f* pX, const
Ipp32f* pY, const Ipp32f* pZ, Ipp32f* pDst[3], Ipp32u N, IppSHState* pSHState);
```

Parameters

pSrcClm	Pointer to the source vector or an array of pointers to the source vectors of pre-computed SHT coefficients of length $(L+1)(L+1)$.
L	Order, can not be greater than maximum order specified in the function `ipprSHInit`.
pX, *pY*, *pZ*	Pointers to the source vectors representing a unit sphere points in Cartesians coordinates.
pDst	Pointer to the destination vector or array pointers to the destination vectors containing the function values reconstructed for each input point.
N	Number of Cartesians points, that is the length of the source vectors *pX*, *pY*, *pZ* and destination vector (or vectors).
pSHState	Pointer to the external buffer for the SHT state structure that must be initialized with maximum order not less than *L*.

Description

The function `ipprSHTInv` is declared in the `ippr.h` file.

All function flavors perform inverse spherical harmonic transform (SHT), that is they reconstruct the function $f(x, y, z)$ defined on a unit sphere by use of its pre-computed SHT coefficients $\{C_l^m : 0 \le l \le L, |m| \le L\}$, that are stored in the source vector or array of vectors *pSrcClm* in order in which they are produced by the function `ipprSHTFwd`.

For each input point $(pX[I], pY[I], pZ[I])$, $0 \le I < N$, it is supposed that $pX[I]^2 + pY[I]^2 + pZ[I]^2 = 1$.

The functions perform the following steps:

1. Compute the SH functions for each input point $\{Y_l^m (pX[I], pY[I], pZ[I]) : 0 \le l \le L, |m| \le L\}$

2. Approximate a unit sphere function $f(x, y, z)$ for each input point by $(pX[I], pY[I], pZ[I])$.

The function `ipprSHTInv_32f_C1` calculates

$$pDst[i] = \sum_{0 \le l \le L,\ |m| \le L} pSrcClm[2l+m] \cdot Y_l^m(pX[i],\ pY[i],\ pZ[i])$$

The function `ipprSHTInv_32f_P3C3` calculates

$$pDst[3i+k] = \sum_{0 \le l \le L,\ |m| \le L} pSrcClm[k][2l+m] \cdot Y_l^m(pX[i],\ pY[i],\ pZ[i])$$

for $0 \le k \le 3$.

The function `ipprSHTInv_32f_P3` calculates

$$pDst[k][i] = \sum_{0 \le l \le L,\ |m| \le L} pSrcClm[k][2l+m] \cdot Y_l^m(pX[i],\ pY[i],\ pZ[i])$$

for $0 \le k \le 3$.

The functions `ipprSHTInv_32f_P3C3` and `ipprSHTInv_32f_P3` are suitable to reconstruct a color function, for example *f:(x, y, z)* --> *R, G, B*, for both pixel-ordered or planar images respectively.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *N* is equal to 0.
`ippStsRangeErr`	Indicates an error condition if *L* is greater that maximum order specified in the function `ipprSHInit`.

Code Example - Simple Tracer

The following code example demonstrates how to use the Intel IPP realistic rendering functions to perform the ray casting of the primary rays.

The input of the tracer is the scene (*Teapot*) specified as two arrays: array of the vertex coordinates (*pTeapotVertCoord*) and array of the indexes of the triangles (*pTeapotIndex*) (see Figure "Structure of Arrays for Triangle Description").

As the result of the tracing, the RGB 3-channel (8u_C3) image is created (see Figure "Rendered Image").

Rendered Image

```
#include "ipp.h"
#define NVERTEX  302
#define NTRIAGLE 576
__declspec (align(16))static const float pTeapotVertCoord[NVERTEX*3] = {
17.500000f, 30.000000f, 0.000000f, 17.303200f, 31.093700f, 0.000000f,
 17.905100f, 31.093800f, 0.000000f, 18.750000f, 30.000000f, 0.000000f,
 15.140700f, 30.000000f, 8.892590f, 14.970500f, 31.093700f, 8.792610f,
 15.491200f, 31.093700f, 9.098440f, 16.222200f, 30.000000f, 9.527780f,
 8.892590f, 30.000000f, 15.140700f, 8.792610f, 31.093700f, 14.970500f,
 9.098440f, 31.093800f, 15.491200f, 9.527780f, 30.000000f, 16.222200f,
 0.000000f, 30.000000f, 17.500000f, 0.000000f, 31.093700f, 17.303200f,
 0.000000f, 31.093800f, 17.905100f, 0.000000f, 30.000000f, 18.750000f,
 -9.392590f, 30.000000f, 15.140700f, -8.940760f, 31.093700f, 14.970500f,
  -9.116960f, 31.093700f, 15.491200f, -9.527780f, 30.000000f, 16.222200f,
 -15.390700f, 30.000000f, 8.892590f, -15.044600f, 31.093700f, 8.792610f,
 -15.500500f, 31.093800f, 9.098440f, -16.222200f, 30.000000f, 9.527780f,
-17.500000f, 30.000000f, 0.000000f, -17.303200f, 31.093700f, 0.000000f,
 -17.905100f, 31.093800f, 0.000000f, -18.750000f, 30.000000f, 0.000000f,
 -15.140700f, 30.000000f, -8.892590f, -14.970500f, 31.093700f, -8.792610f,
```

```
-15.491200f, 31.093700f, -9.098440f, -16.222200f, 30.000000f, -9.527780f,
 -8.892590f, 30.000000f, -15.140700f, -8.792610f, 31.093700f, -14.970500f,
 -9.098440f, 31.093800f, -15.491200f, -9.527780f, 30.000000f, -16.222200f,
0.000000f, 30.000000f, -17.500000f, 0.000000f, 31.093700f, -17.303200f,
 0.000000f, 31.093800f, -17.905100f, 0.000000f, 30.000000f, -18.750000f,
 8.892590f, 30.000000f, -15.140700f, 8.792610f, 31.093700f, -14.970500f,
9.098440f, 31.093700f, -15.491200f, 9.527780f, 30.000000f, -16.222200f,
 15.140700f, 30.000000f, -8.892590f,
14.970500f, 31.093700f, -8.792610f, 15.491200f, 31.093800f, -9.098440f,
 16.222200f, 30.000000f, -9.527780f, 21.759300f, 23.472200f, 0.000000f,
 24.074100f, 17.152800f, 0.000000f,  25.000000f, 11.250000f, 0.000000f,
18.825800f, 23.472200f, 11.056900f, 20.828501f, 17.152800f, 12.233200f,
 21.629601f, 11.250000f, 12.703700f, 11.056900f, 23.472200f, 18.825800f,
 12.233200f, 17.152800f, 20.828501f, 12.703700f, 11.250000f, 21.629601f,
0.000000f, 23.472200f, 21.759300f,  0.000000f, 17.152800f, 24.074100f,
 0.000000f, 11.250000f, 25.000000f,  -11.056900f, 23.472200f, 18.825800f,
 -12.233200f, 17.152800f, 20.828501f, -12.703700f, 11.250000f, 21.629601f,
-18.825800f, 23.472200f, 11.056900f, -20.828501f, 17.152800f, 12.233200f,
 -21.629601f, 11.250000f, 12.703700f, -21.759300f, 23.472200f, 0.000000f,
 -24.074100f, 17.152800f, 0.000000f,  -25.000000f, 11.250000f, 0.000000f,
-18.825800f, 23.472200f, -11.056900f, -20.828501f, 17.152800f, -12.233200f,
 -21.629601f, 11.250000f, -12.703700f, -11.056900f, 23.472200f, -18.825800f,
 -12.233200f, 17.152800f, -20.828501f, -12.703700f, 11.250000f, -21.629601f,
0.000000f, 23.472200f, -21.759300f,  0.000000f, 17.152800f, -24.074100f,
 0.000000f, 11.250000f, -25.000000f,  11.056900f, 23.472200f, -18.825800f,
 12.233200f, 17.152800f, -20.828501f, 12.703700f, 11.250000f, -21.629601f,
 18.825800f, 23.472200f, -11.056900f, 20.828501f, 17.152800f, -12.233200f,
 21.629601f, 11.250000f, -12.703700f, 23.379601f, 6.527780f, 0.000000f,
 20.370399f, 3.472220f, 0.000000f,  18.750000f, 1.875000f, 0.000000f,
  20.227699f, 6.527780f, 11.880300f, 17.624100f, 3.472220f, 10.351200f,
 16.222200f, 1.875000f, 9.527780f,  11.880300f, 6.527780f, 20.227699f,
 10.351200f, 3.472220f, 17.624100f, 9.527780f, 1.875000f, 16.222200f,
 0.000000f,  6.527780f, 23.379601f,  0.000000f, 3.472220f, 20.370399f,
 0.000000f, 1.875000f, 18.750000f, -11.880300f, 6.527780f, 20.227699f,
 -10.351200f, 3.472220f, 17.624100f, -9.527780f, 1.875000f, 16.222200f,
 -20.227699f, 6.527780f, 11.880300f, -17.624100f, 3.472220f, 10.351200f,
 -16.222200f, 1.875000f, 9.527780f,  -23.379601f, 6.527780f, 0.000000f,
 -20.370399f, 3.472220f, 0.000000f, -18.750000f, 1.875000f, 0.000000f,
  -20.227699f, 6.527780f, -11.880300f, -17.624100f, 3.472220f, -10.351200f,
 -16.222200f, 1.875000f, -9.527780f,  -11.880300f, 6.527780f, -20.227699f,
 -10.351200f, 3.472220f, -17.624100f, -9.527780f, 1.875000f, -16.222200f,
  0.000000f, 6.527780f, -23.379601f,  0.000000f, 3.472220f, -20.370399f,
 0.000000f, 1.875000f, -18.750000f,  11.880300f, 6.527780f, -20.227699f,
 10.351200f, 3.472220f, -17.624100f,  9.527780f, 1.875000f, -16.222200f,
  20.227699f, 6.527780f,  -11.880300f, 17.624100f, 3.472220f, -10.351200f,
 16.222200f, 1.875000f,  -9.527780f,  17.847200f, 0.972222f, 0.000000f,
 12.777800f, 0.277778f, 0.000000f, 0.000000f, 0.000000f, 0.000000f, 15.441100f,
 0.972222f, 9.069030f, 11.055100f, 0.277778f, 6.493000f, 9.069030f, 0.972222f,
 15.441100f,  6.493000f, 0.277778f, 11.055100f, 0.000000f, 0.972222f,
 17.847200f, 0.000000f, 0.277778f, 12.777800f, -9.069030f, 0.972222f
, 15.441100f, -6.493000f, 0.277778f, 11.055100f, -15.441100f, 0.972222f,
 9.069030f, -11.055100f, 0.277778f, 6.493000f, -17.847200f, 0.972222f,
 0.000000f, -12.777800f, 0.277778f, 0.000000f, -15.441100f, 0.972222f,
 -9.069030f, -11.055100f, 0.277778f, -6.493000f, -9.069030f, 0.972222f,
 -15.441100f, -6.493000f, 0.277778f, -11.055100f, 0.000000f, 0.972222f,
 -17.847200f, 0.000000f, 0.277778f, -12.777800f, 9.069030f, 0.972222f,
 -15.441100f, 6.493000f, 0.277778f, -11.055100f, 15.441100f, 0.972222f,
 -9.069030f, 11.055100f, 0.277778f, -6.493000f, -20.000000f, 25.312500f,
 0.000000f, -27.453699f, 25.208300f, 0.000000f, -32.129601f, 24.479200f,
 0.000000f, -33.750000f, 22.500000f, 0.000000f, -19.675900f, 26.041700f,
 2.500000f, -27.897800f, 25.910500f, 2.500000f, -32.981800f, 24.992300f,
 2.500000f, -34.722198f, 22.500000f, 2.500000f, -19.074100f, 27.395800f,
```

```
2.500000f, -28.722601f, 27.214500f, 2.500000f, -34.564499f, 25.945200f,
2.500000f, -36.527802f, 22.500000f, 2.500000f, -18.750000f, 28.125000f,
0.000000f, -29.166700f, 27.916700f, 0.000000f, -35.416698f, 26.458300f,
0.000000f, -37.500000f, 22.500000f, 0.000000f, -19.074100f, 27.395800f,
-2.500000f, -28.722601f, 27.214500f, -2.500000f, -34.564499f, 25.945200f,
-2.500000f, -36.527802f, 22.500000f, -2.500000f, -19.675900f, 26.041700f,
-2.500000f, -27.897800f, 25.910500f, -2.500000f, -32.981800f, 24.992300f,
-2.500000f, -34.722198f, 22.500000f, -2.500000f, -32.870399f, 18.958300f,
 0.000000f, -30.046301f, 14.791700f, 0.000000f,  -25.000000f, 11.250000f,
0.000000f, -33.686600f, 18.463200f, 2.500000f, -30.418400f, 14.071500f,
2.500000f,  -24.675900f, 10.277800f, 2.500000f, -35.202301f, 17.543699f,
2.500000f, -31.109400f, 12.734100f, 2.500000f, -24.074100f, 8.472220f,
2.500000f, -36.018501f, 17.048599f, 0.000000f, -31.481501f, 12.013900f,
0.000000f, -23.750000f, 7.500000f, 0.000000f, -35.202301f, 17.543699f,
-2.500000f, -31.109400f, 12.734100f, -2.500000f,  -24.074100f, 8.472220f,
-2.500000f, -33.686600f, 18.463200f, -2.500000f, -30.418400f, 14.071500f,
-2.500000f, -24.675900f, 10.277800f, -2.500000f, 21.250000f, 17.812500f,
0.000000f, 28.379601f, 20.138901f, 0.000000f,  30.787001f, 25.173599f,
0.000000f, 33.750000f, 30.000000f, 0.000000f, 21.250000f, 15.138900f,
5.500000f,  29.243799f, 18.428499f, 4.614200f, 31.867300f, 24.534500f,
2.969140f, 35.694401f, 30.000000f, 2.083330f, 21.250000f, 10.173600f,
5.500000f, 30.848801f, 15.252100f, 4.614200f, 33.873501f, 23.347500f,
2.969140f, 39.305599f, 30.000000f, 2.083330f, 21.250000f, 7.500000f,
0.000000f, 31.712999f, 13.541700f, 0.000000f, 34.953701f, 22.708300f,
0.000000f, 41.250000f, 30.000000f, 0.000000f, 21.250000f, 10.173600f,
-5.500000f, 30.848801f, 15.252100f, -4.614200f, 33.873501f, 23.347500f,
-2.969140f, 39.305599f, 30.000000f, -2.083330f, 21.250000f, 15.138900f
, -5.500000f, 29.243799f, 18.428499f, -4.614200f, 31.867300f, 24.534500f,
-2.969140f, 35.694401f, 30.000000f, -2.083330f, 34.907398f, 30.625000f,
0.000000f, 35.509300f, 30.625000f, 0.000000f,  35.000000f, 30.000000f,
0.000000f, 36.971901f, 30.679001f, 1.867280f,  37.279701f, 30.692499f,
1.466050f, 36.296299f, 30.000000f, 1.250000f, 40.805901f, 30.779301f,
1.867280f, 40.567600f, 30.817900f, 1.466050f, 38.703701f, 30.000000f,
1.250000f, 42.870399f, 30.833300f, 0.000000f, 42.338001f, 30.885401f,
0.000000f, 40.000000f, 30.000000f, 0.000000f, 40.805901f, 30.779301f,
-1.867280f, 40.567600f, 30.817900f, -1.466050f, 38.703701f, 30.000000f,
-1.250000f, 36.971901f, 30.679001f, -1.867280f, 37.279701f, 30.692499f,
-1.466050f, 36.296299f, 30.000000f, -1.250000f, 0.000000f, 39.375000f,
0.000000f, 4.537040f, 38.333302f, 0.000000f, 2.962960f, 36.041698f, 0.000000f,
2.500000f, 33.750000f, 0.000000f, 3.927850f,  38.333302f, 2.310430f,
2.564750f, 36.041698f, 1.508090f, 2.162960f, 33.750000f, 1.270370f, 2.310430f,
38.333302f, 3.927850f,  1.508090f, 36.041698f, 2.564750f, 1.270370f,
33.750000f, 2.162960f, 0.000000f, 38.333302f, 4.537040f, 0.000000f,
36.041698f, 2.962960f, 0.000000f, 33.750000f, 2.500000f, -2.310430f,
38.333302f, 3.927850f, -1.508090f, 36.041698f, 2.564750f, -1.270370f,
33.750000f, 2.162960f, -3.927850f, 38.333302f, 2.310430f, -2.564750f,
36.041698f, 1.508090f, -2.162960f, 33.750000f, 1.270370f, -4.537040f,
38.333302f, 0.000000f, -2.962960f, 36.041698f, 0.000000f, -2.500000f,
33.750000f, 0.000000f, -3.927850f, 38.333302f, -2.310430f,  -2.564750f,
36.041698f, -1.508090f, -2.162960f, 33.750000f, -1.270370f, -2.310430f,
38.333302f, -3.927850f, -1.508090f, 36.041698f, -2.564750f, -1.270370f,
33.750000f, -2.162960f, 0.000000f, 38.333302f, -4.537040f,  0.000000f,
36.041698f, -2.962960f, 0.000000f, 33.750000f, -2.500000f, 2.310430f,
38.333302f, -3.927850f, 1.508090f, 36.041698f, -2.564750f, 1.270370f,
33.750000f, -2.162960f, 3.927850f, 38.333302f, -2.310430f,  2.564750f,
36.041698f, -1.508090f, 2.162960f, 33.750000f, -1.270370f, 7.175930f,
32.361099f, 0.000000f,  13.240700f, 31.388901f, 0.000000f, 16.250000f,
30.000000f, 0.000000f, 6.208500f, 32.361099f, 3.646430f,  11.455700f,
31.388901f, 6.728260f, 14.059300f, 30.000000f, 8.257410f, 3.646430f,
32.361099f, 6.208500f,  6.728260f, 31.388901f, 11.455700f, 8.257410f,
30.000000f, 14.059300f, 0.000000f, 32.361099f, 7.175930f,  0.000000f,
31.388901f, 13.240700f, 0.000000f, 30.000000f, 16.250000f, -3.646430f,
```

```
 32.361099f, 6.208500f, -6.728260f, 31.388901f, 11.455700f, -8.257410f,
 30.000000f, 14.059300f, -6.208500f, 32.361099f, 3.646430f,  -11.455700f,
 31.388901f, 6.728260f, -14.059300f, 30.000000f, 8.257410f, -7.175930f,
 32.361099f, 0.000000f,  -13.240700f, 31.388901f, 0.000000f, -16.250000f,
 30.000000f, 0.000000f, -6.208500f, 32.361099f, -3.646430f,  -11.455700f,
 31.388901f, -6.728260f, -14.059300f, 30.000000f, -8.257410f, -3.646430f,
 32.361099f, -6.208500f, -6.728260f, 31.388901f, -11.455700f, -8.257410f,
 30.000000f, -14.059300f, 0.000000f, 32.361099f, -7.175930f,  0.000000f,
 31.388901f, -13.240700f, 0.000000f, 30.000000f, -16.250000f, 3.646430f,
 32.361099f, -6.208500f,  6.728260f, 31.388901f, -11.455700f, 8.257410f,
 30.000000f, -14.059300f, 6.208500f, 32.361099f, -3.646430f,  11.455700f,
 31.388901f, -6.728260f, 14.059300f, 30.000000f, -8.257410f
};
__declspec (align(16))static const int pTeapotIndex[NTRIAGLE*4] = {
0, 4, 5, 1, 5, 1, 0, 1, 1, 5, 6, 1,
6, 2, 1, 1, 2, 6, 7, 1, 7, 3, 2, 1,
4, 8, 9, 1, 9, 5, 4, 1, 5, 9, 10, 1,
10, 6, 5, 1, 6, 10, 11, 1, 11, 7, 6, 1,
8, 12, 13, 1, 13, 9, 8, 1, 9, 13, 14, 1,
14, 10, 9, 1, 10, 14, 15, 1, 15, 11, 10, 1,
12, 16, 17, 1, 17, 13, 12, 1, 13, 17, 18, 1,
18, 14, 13, 1, 14, 18, 19, 1, 19, 15, 14, 1,
16, 20, 21, 1, 21, 17, 16, 1, 17, 21, 22, 1,
22, 18, 17, 1, 18, 22, 23, 1, 23, 19, 18, 1,
20, 24, 25, 1, 25, 21, 20, 1, 21, 25, 26, 1,
26, 22, 21, 1, 22, 26, 27, 1, 27, 23, 22, 1,
24, 28, 29, 1, 29, 25, 24, 1, 25, 29, 30, 1,
30, 26, 25, 1, 26, 30, 31, 1, 31, 27, 26, 1,
28, 32, 33, 1, 33, 29, 28, 1, 29, 33, 34, 1,
34, 30, 29, 1, 30, 34, 35, 1, 35, 31, 30, 1,
32, 36, 37, 1, 37, 33, 32, 1, 33, 37, 38, 1,
38, 34, 33, 1, 34, 38, 39, 1, 39, 35, 34, 1,
36, 40, 41, 1, 41, 37, 36, 1, 37, 41, 42, 1,
42, 38, 37, 1, 38, 42, 43, 1, 43, 39, 38, 1,
40, 44, 45, 1, 45, 41, 40, 1, 41, 45, 46, 1,
46, 42, 41, 1, 42, 46, 47, 1, 47, 43, 42, 1,
44, 0, 1, 1, 1, 45, 44, 1, 45, 1, 2, 1,
2, 46, 45, 1, 46, 2, 3, 1, 3, 47, 46, 1,
3, 7, 51, 1, 51, 48, 3, 1, 48, 51, 52, 1,
52, 49, 48, 1, 49, 52, 53, 1, 53, 50, 49, 1,
7, 11, 54, 1, 54, 51, 7, 1, 51, 54, 55, 1,
55, 52, 51, 1, 52, 55, 56, 1, 56, 53, 52, 1,
11, 15, 57, 1, 57, 54, 11, 1, 54, 57, 58, 1,
58, 55, 54, 1, 55, 58, 59, 1, 59, 56, 55, 1,
15, 19, 60, 1, 60, 57, 15, 1, 57, 60, 61, 1,
61, 58, 57, 1, 58, 61, 62, 1, 62, 59, 58, 1,
19, 23, 63, 1, 63, 60, 19, 1, 60, 63, 64, 1,
64, 61, 60, 1, 61, 64, 65, 1, 65, 62, 61, 1,
23, 27, 66, 1, 66, 63, 23, 1, 63, 66, 67, 1,
67, 64, 63, 1, 64, 67, 68, 1, 68, 65, 64, 1,
27, 31, 69, 1, 69, 66, 27, 1, 66, 69, 70, 1,
70, 67, 66, 1, 67, 70, 71, 1, 71, 68, 67, 1,
31, 35, 72, 1, 72, 69, 31, 1, 69, 72, 73, 1,
73, 70, 69, 1, 70, 73, 74, 1, 74, 71, 70, 1,
35, 39, 75, 1, 75, 72, 35, 1, 72, 75, 76, 1,
76, 73, 72, 1, 73, 76, 77, 1, 77, 74, 73, 1,
39, 43, 78, 1, 78, 75, 39, 1, 75, 78, 79, 1,
79, 76, 75, 1, 76, 79, 80, 1, 80, 77, 76, 1,
43, 47, 81, 1, 81, 78, 43, 1, 78, 81, 82, 1,
82, 79, 78, 1, 79, 82, 83, 1, 83, 80, 79, 1,
47, 3, 48, 1, 48, 81, 47, 1, 81, 48, 49, 1,
49, 82, 81, 1, 82, 49, 50, 1, 50, 83, 82, 1,
```

```
50, 53, 87, 1, 87, 84, 50, 1, 84, 87, 88, 1,
88, 85, 84, 1, 85, 88, 89, 1, 89, 86, 85, 1,
53, 56, 90, 1, 90, 87, 53, 1, 87, 90, 91, 1,
91, 88, 87, 1, 88, 91, 92, 1, 92, 89, 88, 1,
56, 59, 93, 1, 93, 90, 56, 1, 90, 93, 94, 1,
94, 91, 90, 1, 91, 94, 95, 1, 95, 92, 91, 1,
59, 62, 96, 1, 96, 93, 59, 1, 93, 96, 97, 1,
97, 94, 93, 1, 94, 97, 98, 1, 98, 95, 94, 1,
62, 65, 99, 1, 99, 96, 62, 1, 96, 99, 100, 1,
100, 97, 96, 1, 97, 100, 101, 1, 101, 98, 97, 1,
65, 68, 102, 1, 102, 99, 65, 1, 99, 102, 103, 1,
103, 100, 99, 1, 100, 103, 104, 1, 104, 101, 100, 1,
68, 71, 105, 1, 105, 102, 68, 1, 102, 105, 106, 1,
106, 103, 102, 1, 103, 106, 107, 1, 107, 104, 103, 1,
71, 74, 108, 1, 108, 105, 71, 1, 105, 108, 109, 1,
109, 106, 105, 1, 106, 109, 110, 1, 110, 107, 106, 1,
74, 77, 111, 1, 111, 108, 74, 1, 108, 111, 112, 1,
112, 109, 108, 1, 109, 112, 113, 1, 113, 110, 109, 1,
77, 80, 114, 1, 114, 111, 77, 1, 111, 114, 115, 1,
115, 112, 111, 1, 112, 115, 116, 1, 116, 113, 112, 1,
80, 83, 117, 1, 117, 114, 80, 1, 114, 117, 118, 1,
118, 115, 114, 1, 115, 118, 119, 1, 119, 116, 115, 1,
83, 50, 84, 1, 84, 117, 83, 1, 117, 84, 85, 1,
85, 118, 117, 1, 118, 85, 86, 1, 86, 119, 118, 1,
86, 89, 123, 1, 123, 120, 86, 1, 120, 123, 124, 1,
124, 121, 120, 1, 121, 124, 122, 1, 122, 122, 121, 1,
89, 92, 125, 1, 125, 123, 89, 1, 123, 125, 126, 1,
126, 124, 123, 1, 124, 126, 122, 1, 122, 122, 124, 1,
92, 95, 127, 1, 127, 125, 92, 1, 125, 127, 128, 1,
128, 126, 125, 1, 126, 128, 122, 1, 122, 122, 126, 1,
95, 98, 129, 1, 129, 127, 95, 1, 127, 129, 130, 1,
130, 128, 127, 1, 128, 130, 122, 1, 122, 122, 128, 1,
98, 101, 131, 1, 131, 129, 98, 1, 129, 131, 132, 1,
132, 130, 129, 1, 130, 132, 122, 1, 122, 122, 130, 1,
101, 104, 133, 1, 133, 131, 101, 1, 131, 133, 134, 1,
134, 132, 131, 1, 132, 134, 122, 1, 122, 122, 132, 1,
104, 107, 135, 1, 135, 133, 104, 1, 133, 135, 136, 1,
136, 134, 133, 1, 134, 136, 122, 1, 122, 122, 134, 1,
107, 110, 137, 1, 137, 135, 107, 1, 135, 137, 138, 1,
138, 136, 135, 1, 136, 138, 122, 1, 122, 122, 136, 1,
110, 113, 139, 1, 139, 137, 110, 1, 137, 139, 140, 1,
140, 138, 137, 1, 138, 140, 122, 1, 122, 122, 138, 1,
113, 116, 141, 1, 141, 139, 113, 1, 139, 141, 142, 1,
142, 140, 139, 1, 140, 142, 122, 1, 122, 122, 140, 1,
116, 119, 143, 1, 143, 141, 116, 1, 141, 143, 144, 1,
144, 142, 141, 1, 142, 144, 122, 1, 122, 122, 142, 1,
119, 86, 120, 1, 120, 143, 119, 1, 143, 120, 121, 1,
121, 144, 143, 1, 144, 121, 122, 1, 122, 122, 144, 1,
145, 149, 150, 1, 150, 146, 145, 1, 146, 150, 151, 1,
151, 147, 146, 1, 147, 151, 152, 1, 152, 148, 147, 1,
149, 153, 154, 1, 154, 150, 149, 1, 150, 154, 155, 1,
155, 151, 150, 1, 151, 155, 156, 1, 156, 152, 151, 1,
153, 157, 158, 1, 158, 154, 153, 1, 154, 158, 159, 1,
159, 155, 154, 1, 155, 159, 160, 1, 160, 156, 155, 1,
157, 161, 162, 1, 162, 158, 157, 1, 158, 162, 163, 1,
163, 159, 158, 1, 159, 163, 164, 1, 164, 160, 159, 1,
161, 165, 166, 1, 166, 162, 161, 1, 162, 166, 167, 1,
167, 163, 162, 1, 163, 167, 168, 1, 168, 164, 163, 1,
165, 145, 146, 1, 146, 166, 165, 1, 166, 146, 147, 1,
147, 167, 166, 1, 167, 147, 148, 1, 148, 168, 167, 1,
148, 152, 172, 1, 172, 169, 148, 1, 169, 172, 173, 1,
173, 170, 169, 1, 170, 173, 174, 1, 174, 171, 170, 1,
```

```
152, 156, 175, 1, 175, 172, 152, 1, 172, 175, 176, 1,
176, 173, 172, 1, 173, 176, 177, 1, 177, 174, 173, 1,
156, 160, 178, 1, 178, 175, 156, 1, 175, 178, 179, 1,
179, 176, 175, 1, 176, 179, 180, 1, 180, 177, 176, 1,
160, 164, 181, 1, 181, 178, 160, 1, 178, 181, 182, 1,
182, 179, 178, 1, 179, 182, 183, 1, 183, 180, 179, 1,
164, 168, 184, 1, 184, 181, 164, 1, 181, 184, 185, 1,
185, 182, 181, 1, 182, 185, 186, 1, 186, 183, 182, 1,
168, 148, 169, 1, 169, 184, 168, 1, 184, 169, 170, 1,
170, 185, 184, 1, 185, 170, 171, 1, 171, 186, 185, 1,
187, 191, 192, 1, 192, 188, 187, 1, 188, 192, 193, 1,
193, 189, 188, 1, 189, 193, 194, 1, 194, 190, 189, 1,
191, 195, 196, 1, 196, 192, 191, 1, 192, 196, 197, 1,
197, 193, 192, 1, 193, 197, 198, 1, 198, 194, 193, 1,
195, 199, 200, 1, 200, 196, 195, 1, 196, 200, 201, 1,
201, 197, 196, 1, 197, 201, 202, 1, 202, 198, 197, 1,
199, 203, 204, 1, 204, 200, 199, 1, 200, 204, 205, 1,
205, 201, 200, 1, 201, 205, 206, 1, 206, 202, 201, 1,
203, 207, 208, 1, 208, 204, 203, 1, 204, 208, 209, 1,
209, 205, 204, 1, 205, 209, 210, 1, 210, 206, 205, 1,
207, 187, 188, 1, 188, 208, 207, 1, 208, 188, 189, 1,
189, 209, 208, 1, 209, 189, 190, 1, 190, 210, 209, 1,
190, 194, 214, 1, 214, 211, 190, 1, 211, 214, 215, 1,
215, 212, 211, 1, 212, 215, 216, 1, 216, 213, 212, 1,
194, 198, 217, 1, 217, 214, 194, 1, 214, 217, 218, 1,
218, 215, 214, 1, 215, 218, 219, 1, 219, 216, 215, 1,
198, 202, 220, 1, 220, 217, 198, 1, 217, 220, 221, 1,
221, 218, 217, 1, 218, 221, 222, 1, 222, 219, 218, 1,
202, 206, 223, 1, 223, 220, 202, 1, 220, 223, 224, 1,
224, 221, 220, 1, 221, 224, 225, 1, 225, 222, 221, 1,
206, 210, 226, 1, 226, 223, 206, 1, 223, 226, 227, 1,
227, 224, 223, 1, 224, 227, 228, 1, 228, 225, 224, 1,
210, 190, 211, 1, 211, 226, 210, 1, 226, 211, 212, 1,
212, 227, 226, 1, 227, 212, 213, 1, 213, 228, 227, 1,
229, 229, 233, 1, 233, 230, 229, 1, 230, 233, 234, 1,
234, 231, 230, 1, 231, 234, 235, 1, 235, 232, 231, 1,
229, 229, 236, 1, 236, 233, 229, 1, 233, 236, 237, 1,
237, 234, 233, 1, 234, 237, 238, 1, 238, 235, 234, 1,
229, 229, 239, 1, 239, 236, 229, 1, 236, 239, 240, 1,
240, 237, 236, 1, 237, 240, 241, 1, 241, 238, 237, 1,
229, 229, 242, 1, 242, 239, 229, 1, 239, 242, 243, 1,
243, 240, 239, 1, 240, 243, 244, 1, 244, 241, 240, 1,
229, 229, 245, 1, 245, 242, 229, 1, 242, 245, 246, 1,
246, 243, 242, 1, 243, 246, 247, 1, 247, 244, 243, 1,
229, 229, 248, 1, 248, 245, 229, 1, 245, 248, 249, 1,
249, 246, 245, 1, 246, 249, 250, 1, 250, 247, 246, 1,
229, 229, 251, 1, 251, 248, 229, 1, 248, 251, 252, 1,
252, 249, 248, 1, 249, 252, 253, 1, 253, 250, 249, 1,
229, 229, 254, 1, 254, 251, 229, 1, 251, 254, 255, 1,
255, 252, 251, 1, 252, 255, 256, 1, 256, 253, 252, 1,
229, 229, 257, 1, 257, 254, 229, 1, 254, 257, 258, 1,
258, 255, 254, 1, 255, 258, 259, 1, 259, 256, 255, 1,
229, 229, 260, 1, 260, 257, 229, 1, 257, 260, 261, 1,
261, 258, 257, 1, 258, 261, 262, 1, 262, 259, 258, 1,
229, 229, 263, 1, 263, 260, 229, 1, 260, 263, 264, 1,
264, 261, 260, 1, 261, 264, 265, 1, 265, 262, 261, 1,
229, 229, 230, 1, 230, 263, 229, 1, 263, 230, 231, 1,
231, 264, 263, 1, 264, 231, 232, 1, 232, 265, 264, 1,
232, 235, 269, 1, 269, 266, 232, 1, 266, 269, 270, 1,
270, 267, 266, 1, 267, 270, 271, 1, 271, 268, 267, 1,
235, 238, 272, 1, 272, 269, 235, 1, 269, 272, 273, 1,
273, 270, 269, 1, 270, 273, 274, 1, 274, 271, 270, 1,
```

```
238, 241, 275, 1, 275, 272, 238, 1, 272, 275, 276, 1,
276, 273, 272, 1, 273, 276, 277, 1, 277, 274, 273, 1,
241, 244, 278, 1, 278, 275, 241, 1, 275, 278, 279, 1,
279, 276, 275, 1, 276, 279, 280, 1, 280, 277, 276, 1,
244, 247, 281, 1, 281, 278, 244, 1, 278, 281, 282, 1,
282, 279, 278, 1, 279, 282, 283, 1, 283, 280, 279, 1,
247, 250, 284, 1, 284, 281, 247, 1, 281, 284, 285, 1,
285, 282, 281, 1, 282, 285, 286, 1, 286, 283, 282, 1,
250, 253, 287, 1, 287, 284, 250, 1, 284, 287, 288, 1,
288, 285, 284, 1, 285, 288, 289, 1, 289, 286, 285, 1,
253, 256, 290, 1, 290, 287, 253, 1, 287, 290, 291, 1,
291, 288, 287, 1, 288, 291, 292, 1, 292, 289, 288, 1,
256, 259, 293, 1, 293, 290, 256, 1, 290, 293, 294, 1,
294, 291, 290, 1, 291, 294, 295, 1, 295, 292, 291, 1,
259, 262, 296, 1, 296, 293, 259, 1, 293, 296, 297, 1,
297, 294, 293, 1, 294, 297, 298, 1, 298, 295, 294, 1,
262, 265, 299, 1, 299, 296, 262, 1, 296, 299, 300, 1,
300, 297, 296, 1, 297, 300, 301, 1, 301, 298, 297, 1,
265, 232, 266, 1, 266, 299, 265, 1, 299, 266, 267, 1,
267, 300, 299, 1, 300, 267, 268, 1, 268, 301, 300, 1
};
static void rrOptimCheckMask_32s( Ipp32s* pMask, int len, int* check)
{
    Ipp32s pMax=-1;
    if( pMask[0] != -1 )
    {
        *check = 33;
        return;
    }
    if( pMask[len-1] != -1 )
    {
        *check = 33;
        return;
    }
    ippsMax_32s( pMask, len, &pMax );
    *check = pMax;
    return;
}


#define MAX_KDTREE_DEPTH 33

#define _WIDTH  320
#define _HEIGHT 256
typedef struct  SceneContext {
    int                   nVert;
    Ipp32f                *pVrt;        /* pointer to array of Vertices */
    int               nTriangles;
    Ipp32s                *pTng;        /* pointer to array of Triangels */
    IpprIntersectContext    iCtx;        /* Intersection context */
}SampleSceneContext;

#define NBUNCH 10
typedef struct  TraceContext {
    Ipp32f      *p3D_0[3];
    Ipp32f      *p1D_1   ;
    Ipp32f      *p2D_2[2];
    Ipp32s      *pTrngl;
    Ipp32f      *p3D_3[3];
}SampleTraceContext;
```

```
int main(/*int argc, char** argv*/)
{
    IppiSize  sizeImPlane={_WIDTH,_HEIGHT}; /* pixels */
    SampleSceneContext sceneContext;
    IppStatus          status;
    Ipp32f            *pFlatNorm;
    IppBox3D_32f       pBound;
    int pTriAccelSize;
    int KDTreeSize;
    IppPoint3D_32f  ul_corner = {-0.70281667f,    0.52656168f,  -1.000000f},
                                          /* upper left coner of Image plane */
                    dx        = { 0.0044063739f,  0.000000f,     0.000000f},
                    dy        = { 0.000000f,     -0.0044063739f, 0.000000f},
                    eye_pos   = { 2.6851997f,     23.162521f,   75.012863f};
    int isNotFinished = 1;
    IppiSize bunchSize = { 16, 16};
    int lenBunch=bunchSize.height *bunchSize.width;
    int nBunch = NBUNCH; /* #bunch of rays  */
    SampleTraceContext rtContext;
    Ipp32f* pMemForTrace;
    int pStepBMP;
    Ipp8u* pBMP;
    Ipp8u value[3];
    IppPoint3D_32f defColour;
    int xCnt;
    int yCnt;
    int last_cnt;
    int stepBunch;
    int counter;
    IpprPSAHBldContext fastKDCont;
    sceneContext.nVert      = NVERTEX;
    sceneContext.nTriangles = NTRIAGLE;
    /* allocate memory for the coordinates of triangle's vertexes. */
    sceneContext.pVrt = ippsMalloc_32f( sceneContext.nVert * 3 );
    if (!sceneContext.pVrt )  return -1;

    status = ippsCopy_32f( pTeapotVertCoord, sceneContext.pVrt,
                             sceneContext.nVert * 3 );
    if(status < 0 ) return -1;
    /* allocate memory for the triangle's indexes */
    sceneContext.pTng = ippsMalloc_32s( sceneContext.nTriangles * 4 );
    if (!sceneContext.pVrt ) return -1;

    status = ippsCopy_32s( pTeapotIndex, sceneContext.pTng,
                              sceneContext.nTriangles * 4);
    if(status < 0 ) return -1;
    /* allocate memory for the triangle's flat normales */
    pFlatNorm = ippsMalloc_32f( sceneContext.nTriangles * 3 );
    ipprTriangleNormal_32f(sceneContext.pVrt, sceneContext.pTng, pFlatNorm,
                              sceneContext.nTriangles);
    /*
    /////////////////////////////////////////////////////
    // Initialization of the acceleration structures ////
    /////////////////////////////////////////////////////
    */
    /* create the AABB  */
    status = ipprSetBoundBox_32f( sceneContext.pVrt, sceneContext.nVert, &(pBound));
    if(status < 0 )  return -1;
    sceneContext.iCtx.pBound = &(pBound);
    /* create triangle acceleration structure */
    ipprTriangleAccelGetSize( &pTriAccelSize);
    sceneContext.iCtx.pAccel = (IpprTriangleAccel *)ippsMalloc_8u( pTriAccelSize *
```

```
                                sceneContext.nTriangles );
status = ipprTriangleAccelInit( sceneContext.iCtx.pAccel, sceneContext.pVrt,
                                    sceneContext.pTng, sceneContext.nTriangles );
if(status < 0 ) return -1;

/* create the KDtree structure */
fastKDCont.Bounds = 0;
fastKDCont.Alg = ippKDTBuildPureSAH;
fastKDCont.AvailMemory = 2046;
fastKDCont.MaxDepth = MAX_KDTREE_DEPTH;
fastKDCont.QoS = 1.0f;

status = ipprKDTreeBuildAlloc(
   &(sceneContext.iCtx.pRootNode),
   sceneContext.pVrt,
   sceneContext.pTng,
   sceneContext.nVert,
   sceneContext.nTriangles,
   &KDTreeSize,
   (void*)&fastKDCont );
if(status < 0 ) return -1;

/* allocate memory for all temporary arrays */
pMemForTrace = ippsMalloc_32f( bunchSize.height *bunchSize.width * nBunch );
if(!pMemForTrace) return -1;
rtContext.p3D_0[0] = pMemForTrace + 0*lenBunch;
rtContext.p3D_0[1] = pMemForTrace + 1*lenBunch;
rtContext.p3D_0[2] = pMemForTrace + 2*lenBunch;
rtContext.p1D_1    = pMemForTrace + 3*lenBunch;
rtContext.p2D_2[0] = pMemForTrace + 4*lenBunch;
rtContext.p2D_2[1] = pMemForTrace + 5*lenBunch;
rtContext.pTrngl =(Ipp32s*)(pMemForTrace + 6*lenBunch);
rtContext.p3D_3[0] = pMemForTrace + 7*lenBunch;
rtContext.p3D_3[1] = pMemForTrace + 8*lenBunch;
rtContext.p3D_3[2] = pMemForTrace + 9*lenBunch;
/* allocate memory for resal Image */
pBMP = ippiMalloc_8u_C3(sizeImPlane.width, sizeImPlane.height, &pStepBMP);
if(!pBMP) return -1;
/* init resal Image */
 value[0] =value[1] =value[2] = 0;
ippiSet_8u_C3R( value, pBMP, pStepBMP, sizeImPlane );
/* for simplicity we'll   */
xCnt      = sizeImPlane.width  / bunchSize.width;
yCnt      = sizeImPlane.height / bunchSize.height;
last_cnt  = xCnt * yCnt;
stepBunch = bunchSize.width*sizeof(float);
counter   = 0;
defColour[0] = 0.999f;
defColour[1] = 0.899f;
defColour[2] = 0.799f;
/*
   Simple tracer
   Groups of rays, or bunches, of the size bunchSize are traced.
*/
while(isNotFinished)
{
    int x = counter % xCnt, y = counter / xCnt;
    int check;
    Ipp8u* pCurrentBMP_8u;
    Ipp8u* pBMP_8u[3];
    pCurrentBMP_8u = pBMP + bunchSize.width * x * 3 +
                              bunchSize.height * y * pStepBMP;
```

```
        /* calculate Eye/Primary Rays */
        ipprCastEye_32f(
            ul_corner,
            dx, dy,
            x, y, /* coordinates of the bunch on the Image playne */
            bunchSize,
            rtContext.p3D_0, /* direction of primary rays */
            bunchSize);
        ippsSet_32f(IPP_MAXABS_32F, rtContext.p1D_1, lenBunch);

        ipprIntersectEyeSO_32f(eye_pos,
            rtContext.p3D_0,    /* direction of primary rays                     */
            rtContext.p1D_1,    /* distance from origin to intersection point.   */
            rtContext.p2D_2,    /* local surface parameters( u, v )at hit point. */
            rtContext.pTrngl,   /* the Triangle index, just it's array of masks  */
            &sceneContext.iCtx, bunchSize);
        rrOptimCheckMask_32s( rtContext.pTrngl, lenBunch, &check);
        if( check== -1 ){
            goto secondaryEnd; /* there was no any intersections */
        }
        /* simple shader is based on Lambert low */
        ipprSurfFlatNormal_32f(pFlatNorm, rtContext.pTrngl,
            rtContext.p3D_3,
            lenBunch);
        ipprDot_32f_P3C1M( rtContext.p3D_0, rtContext.p3D_3, rtContext.pTrngl,
            rtContext.p1D_1, /* < N * EyeDir > */
            lenBunch);
        ippsAbs_32f_I( rtContext.p1D_1, lenBunch);
        ippsSet_32f( defColour[0], rtContext.p3D_0[0], lenBunch );
        ippsSet_32f( defColour[1], rtContext.p3D_0[1], lenBunch );
        ippsSet_32f( defColour[2], rtContext.p3D_0[2], lenBunch );
        ipprMul_32f_C1P3IM( rtContext.p1D_1, rtContext.pTrngl, rtContext.p3D_0,
                               lenBunch);

        pBMP_8u[0] = (Ipp8u*)rtContext.p3D_3[0];
        pBMP_8u[1] = (Ipp8u*)rtContext.p3D_3[1];
        pBMP_8u[2] = (Ipp8u*)rtContext.p3D_3[2];

        ippiReduceBits_32f8u_C1R( rtContext.p3D_0[0], stepBunch, pBMP_8u[0],
                                     stepBunch, bunchSize, 0, ippDitherBayer, 255);
        ippiReduceBits_32f8u_C1R( rtContext.p3D_0[1], stepBunch, pBMP_8u[1],
                                     stepBunch, bunchSize, 0, ippDitherBayer, 255);
        ippiReduceBits_32f8u_C1R( rtContext.p3D_0[2], stepBunch, pBMP_8u[2],
                                     stepBunch, bunchSize, 0, ippDitherBayer, 255);
        ippiCopy_8u_P3C3R( pBMP_8u, stepBunch, pCurrentBMP_8u, pStepBMP,
                             bunchSize );
secondaryEnd:
        if(++counter==last_cnt) isNotFinished = 0;
    }
    /* free context */
    ippsFree(sceneContext.pVrt);
    ippsFree(sceneContext.pTng);
    ippsFree(sceneContext.iCtx.pAccel);
    ippsFree(pMemForTrace);
    ippsFree(pFlatNorm);
    ippiFree(pBMP);
    ipprKDTreeFree(sceneContext.iCtx.pRootNode);
    return 0;
}
```

3D Data Transforms Functions

The functions described in this section perform 3D transforms - resizing, affine transform, and remapping.

ResizeGetBufSize

Calculates the size of the external work buffer for the function `ipprResize`*.*

Syntax

`IppStatus ipprResizeGetBufSize(IpprCuboid srcVoi, IpprCuboid dstVoi, int nChannel, int interpolation, int* pSize);`

Parameters

srcVoi	Volume of interest in the source volume.
dstVoi	Volume of interest in the destination volume.
nChannel	Number of channels, possible value: 1.
interpolation	Type of interpolation, the following values are possible: IPPI_INTER_NN - nearest neighbor interpolation, IPPI_INTER_LINEAR - trilinear interpolation, IPPI_INTER_CUBIC - tricubic interpolation, IPPI_INTER_CUBIC2P_BSPLINE - B-spline, IPPI_INTER_CUBIC2P_CATMULLROM - Catmull-Rom spline, IPPI_INTER_CUBIC2P_B05C03 - special two-parameters filter (1/2, 3/10).
pSize	Pointer to the size of the external buffer.

Description

The function `ipprResizeGetBufSize` is declared in the `ippr.h` file. This function calculates the size of the external buffer required for the functions `ipprResize`.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error.
ippStsNullPtrErr	Indicates an error if *pSize* pointer is NULL.
ippStsSizeErr	Indicates an error if width, or height, or depth of the *srcVoi* or *dstVoi* is less than or equal to 0.
ippStsNumChannelErr	Indicates an error condition if *nChannel* is not equal to 1.
ippStsInterpolationErr	Indicates an error condition if *interpolation* has an illegal value.

Resize

Resizes the source volume.

Syntax

```
IppStatus ipprResize_8u_C1V(const Ipp8u* pSrc, IpprVolume srcVolume, int srcStep, int
srcPlaneStep, IpprCuboid srcVoi, Ipp8u* pDst, int dstStep, int dstPlaneStep, IpprCuboid
dstVoi, double xFactor, double yFactor, double zFactor, double xShift, double yShift,
double zShift, int interpolation, Ipp8u* pBuffer);
```

```
IppStatus ipprResize_16u_C1V(const Ipp16u* pSrc, IpprVolume srcVolume, int srcStep, int
srcPlaneStep, IpprCuboid srcVoi, Ipp16u* pDst, int dstStep, int dstPlaneStep, IpprCuboid
dstVoi, double xFactor, double yFactor, double zFactor, double xShift, double yShift,
double zShift, int interpolation, Ipp8u* pBuffer);
```

```
IppStatus ipprResize_8u_C1PV(const Ipp8u* const pSrc[], IpprVolume srcVolume, int srcStep,
IpprCuboid srcVoi, Ipp8u* const pDst[], int dstStep, IpprCuboid dstVoi, double xFactor,
double yFactor, double zFactor, double xShift, double yShift, double zShift, int
interpolation, Ipp8u* pBuffer);
```

```
IppStatus ipprResize_16u_C1PV(const Ipp16u* const pSrc[], IpprVolume srcVolume, int srcStep,
IpprCuboid srcVoi, Ipp16u* const pDst[], int dstStep, IpprCuboid dstVoi, double xFactor,
double yFactor, double zFactor, double xShift, double yShift, double zShift, int
interpolation, Ipp8u* pBuffer);
```

Parameters

pSrc	Pointer to the source volume origin. An array of pointers to the source planes for non-contiguous volume.
srcVolume	Size of the source volume.
srcStep	Distance in bytes between starts of consecutive lines in each plane of the source volume.
srcPlaneStep	Distance in bytes between planes of the source contiguous volume.
srcVoi	Volume of interest of the source volume.
pDst	Pointer to the destination volume origin. An array of pointers to the destination planes for non-contiguous volume.
dstStep	Distance in bytes between starts of consecutive lines in each plane of the the destination volume.
dstPlaneStep	Distance in bytes between planes of the destination contiguous volume.
dstVoi	Volume of interest of the destination volume.
x-, y-, zFactor	Factors by which the *x*, *y*, z dimensions of the source VOI are changed.
x-, y-, zShift	Shift values in the *x*, *y*, and *z* directions respectively.

interpolation	Type of interpolation, the following values are possible: `IPPI_INTER_NN` - nearest neighbor interpolation, `IPPI_INTER_LINEAR` - trilinear interpolation, `IPPI_INTER_CUBIC` - tricubic interpolation, `IPPI_INTER_CUBIC2P_BSPLINE` - B-spline, `IPPI_INTER_CUBIC2P_CATMULLROM` - Catmull-Rom spline, `IPPI_INTER_CUBIC2P_B05C03` - special two-parameters filter (1/2, 3/10).
pBuffer	Pointer to the external buffer.

Description

The function `ipprResize` is declared in the `ippr.h` file. It operates with volume of interest (VOI).

This function resizes the source volume *srcVoi* by *xFactor* in the *x* direction, *yFactor* in the *y* direction and *zFactor* in the *z* direction. The volume size can be reduced or increased in each direction, depending on the values of *xFactor, yFactor, zFactor*. If the value of the certain factor is greater than 1, the volume size is increased, and if it is less than 1, the volume size is reduced in the corresponding direction. The result is resampled using the interpolation method specified by the *interpolation* parameter, and written to the destination volume VOI.

Coordinates *x'*, *y'*, and *z'* in the resized volume are obtained from the equations:

$x' = xFactor * x + xShift$

$y' = yFactor * x + yShift$

$z' = zFactor * z + zShift$

where *x*, *y*, and *z* denote the coordinates of the element in the source volume. The right coordinate system (RCS) is used here.

The function requires the external buffer *pBuffer*, its size must be previously computed by calling the function ipprResizeGetBufSize.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if width, or height, or depth of the source and destination volumes is less than or equal to 0.
`ippStsResizeFactorErr`	Indicates an error condition if one of the *xFactor*, *yFactor*, *zFactor* is less than or equal to 0.
`ippStsInterpolationErr`	Indicates an error condition if *interpolation* has an illegal value.
`ippStsWrongIntersectVOI`	Indicates a warning if *srcVoi* has not intersection with the source volume, operation is not performed.

GetResizeCuboid

Computes coordinates of the destination cuboid.

Syntax

```
IppStatusipprGetResizeCuboid(IpprCuboid srcVoi, Ipp8u* , IpprCuboid* pDstCuboid, double
xFactor, double yFactor, double zFactor, double xShift, double yShift, double zShift, int
interpolation);
```

Parameters

srcVoi	Volume of interest of the source volume.
pDstCuboid	Pointer to the destination cuboid.
x-, y-, zFactor	Factors by which the *x*, *y*, z dimensions of the source VOI are changed.
x-, y-, zShift	Shift values in the *x*, *y*, and *z* directions respectively.
interpolation	Type of interpolation, the following values are possible: IPPI_INTER_NN - nearest neighbor interpolation, IPPI_INTER_LINEAR - trilinear interpolation, IPPI_INTER_CUBIC - tricubic interpolation, IPPI_INTER_CUBIC2P_BSPLINE - B-spline, IPPI_INTER_CUBIC2P_CATMULLROM - Catmull-Rom spline, IPPI_INTER_CUBIC2P_B05C03 - special two-parameters filter (1/2, 3/10).

Description

The function `ipprGetResizeCuboid` is declared in the `ippr.h` file. It operates with volume of interest (VOI).

This function computes the coordinates of the resultant cubouid which is obtained if the source volume *srcVoi* is resized with the specified parameters. The resize operation is not performed.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error if the pointer *pDstCuboid* is `NULL`.
`ippStsSizeErr`	Indicates an error if width, or height, or depth of the source and destination volumes is less than or equal to 0.
`ippStsResizeFactorErr`	Indicates an error condition if one of the *xFactor*, *yFactor*, *zFactor* is less than or equal to 0.
`ippStsInterpolationErr`	Indicates an error condition if *interpolation* has an illegal value.

WarpAffineGetBufSize

Calculates the size of the external buffer for the affine transform.

Syntax

```
IppStatus npprWarpAffineGetBufSize(IpprCuboid srcVoi, IpprCuboid dstVoi, int nChannel,
int interpolation, int* pBufferSize);
```

Parameters

srcVoi	Volume of interest of the source volume.
dstVoi	Volume of interest of the destination volume.
nChannel	Number of channel or planes, possible value is 1.
interpolation	Type of interpolation, the following values are possible: IPPI_INTER_NN - nearest neighbor interpolation, IPPI_INTER_LINEAR - trilinear interpolation, IPPI_INTER_CUBIC - tricubic interpolation, IPPI_INTER_CUBIC2P_BSPLINE - B-spline, IPPI_INTER_CUBIC2P_CATMULLROM - Catmull-Rom spline, IPPI_INTER_CUBIC2P_B05C03 - special two-parameters filter (1/2, 3/10).
pBufferSize	Pointer to the size of the external buffer.

Description

The function `npprWarpAffineGetBufSize` is declared in the `ippr.h` file.

This function calculates the size (in bytes) of the external buffer that is required for the function npprWarpAffine npprWarpAffine. (In some cases the function returns zero size of the buffer).

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or a warning.
ippStsNullPtrErr	Indicates an error if othe *pBufferSize* pointer is NULL.
ippStsSizeErr	Indicates an error if width, or height, or depth of the *srcVoi* or *dstVoi* is less than or equal to 0.
ippStsNumChannelErr	Indicates an error condition if *nChannel* has an illegal value.
ippStsInterpolationErr	Indicates an error condition if *interpolation* has an illegal value.

WarpAffine

Performs the general affine transform of the source volume.

Syntax

```
IppStatusypprWarpAffine_8u_C1PV(const Ipp8u* const pSrc[], IpprVolume srcVolume, int
srcStep, IpprCuboid srcVoi, Ipp8u* const pDst[], int dstStep, IpprCuboid dstVoi, const
double coeffs[3][4], int interpolation, Ipp8u* pBuffer);
```

```
IppStatus ipprWarpAffine_16u_C1PV(const Ipp16u* const pSrc[], IpprVolume srcVolume, int
srcStep, IpprCuboid srcVoi, Ipp16u* const pDst[], int dstStep, IpprCuboid dstVoi, const
double coeffs[3][4], int interpolation, Ipp8u* pBuffer);
```

Parameters

pSrc	Array of pointers to the planes in the source volume.
srcVolume	Size in pixels of the source volume.
srcStep	Distance in bytes between starts of consecutive lines in each plane of the source volume.
srcVoi	Volume of interest of the source volume.
pDst	Array of pointers to the planes in the destination volume.
dstVoi	Volume of interest of the destination volume.
coeffs	Coefficients of the affine transform.
interpolation	Type of interpolation, the following values are possible: IPPI_INTER_NN - nearest neighbor interpolation, IPPI_INTER_LINEAR - trilinear interpolation, IPPI_INTER_CUBIC - tricubic interpolation, IPPI_INTER_CUBIC2P_BSPLINE - B-spline, IPPI_INTER_CUBIC2P_CATMULLROM - Catmull-Rom spline, IPPI_INTER_CUBIC2P_B05C03 - special two-parameters filter (1/2, 3/10).
pBuffer	Pointer to the external buffer.

Description

The function `ipprResize` is declared in the `ippr.h` file. It operates with volume of interest (VOI).

This affine warp function transforms the coordinates (x, y, z) of the source volume voxels according to the following formulas:

$x' = c_{00} * x + c_{01} * y + c_{02} * z + c_{03}$

$y' = c_{10} * x + c_{11} * y + c_{12} * z + c_{13}$

$z' = c_{20} * x + c_{21} * y + c_{22} * z + c_{23}$

where *x'*, *y'* and *z'* denote the voxel coordinates in the transformed volume, and c_{ij} are the affine transform coefficients stored in the array *coeffs*.

The function requires the external buffer *pBuffer*, its size can be previously computed by calling the function ipprWarpAffineGetBufSize.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or a warning.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if width, or height, or depth of the source and destination volumes is less than or equal to 0.
ippStsCoeffErr	Indicates an error condition if determinant of the transform matrix c_{ij} is equal to 0.
ippStsInterpolationErr	Indicates an error condition if *interpolation* has an illegal value.
ippStsWrongIntersectVOI	Indicates a warning if *srcVoi* has not intersection with the source volume, operation is not performed.

Remap

Performs the look-up coordinate mapping of the elements of the source volume.

Syntax

Operation on non-contiguous volume data

```
IppStatus ipprRemap_8u_C1PV(const Ipp8u* const pSrc[], IpprVolume srcVolume, int srcStep,
IpprCuboid srcVoi, const Ipp32f* const pxMap[], const Ipp32f* const pyMap[], const Ipp32f*
const pzMap[], int mapStep, Ipp8u* const pDst[], int dstStep, IpprVolume dstVolume, int
interpolation);
```

```
IppStatus ipprRemap_16u_C1PV(const Ipp16u* const pSrc[], IpprVolume srcVolume, int srcStep,
IpprCuboid srcVoi, const Ipp32f* const pxMap[], const Ipp32f* const pyMap[], const Ipp32f*
const pzMap[], int mapStep, Ipp16u* const pDst[], int dstStep, IpprVolume dstVolume, int
interpolation);
```

```
IppStatus ipprRemap_32f_C1PV(const Ipp32f* const pSrc[], IpprVolume srcVolume, int srcStep,
IpprCuboid srcVoi, const Ipp32f* const pxMap[], const Ipp32f* const pyMap[], const Ipp32f*
const pzMap[], int mapStep, Ipp32f* const pDst[], int dstStep, IpprVolume dstVolume, int
interpolation);
```

Parameters

pSrc	Array of pointers to the planes in the source volume.
srcVolume	Size of the source volume.
srcStep	Distance in bytes between starts of consecutive lines in every plane of the source volume.
srcVoi	Region of interest in the source volume.

pxMap, *pyMap*, *pzMap*	Arrays of pointers to the starts of the 2D buffers, containing tables of the x-, y- and z-coordinates. If the referenced coordinates correspond to a voxel outside of the source VOI, no mapping of the source pixel is done.
mapStep	Step in bytes through the buffers containing tables of the x-, y- and z-coordinates.
pDst	Array of the pointers to the planes in the destination volume.
dstStep	Distance in bytes between starts of consecutive lines in every plane of the destination volume.
dstVolume	Size of the destination volume.
interpolation	The type of interpolation, the following values are possible: `IPPI_INTER_NN` - nearest neighbor interpolation, `IPPI_INTER_LINEAR` - trilinear interpolation, `IPPI_INTER_CUBIC` - tricubic interpolation, `IPPI_INTER_CUBIC2P_BSPLINE` - B-spline, `IPPI_INTER_CUBIC2P_CATMULLROM` - Catmull-Rom spline, `IPPI_INTER_CUBIC2P_B05C03` - special two-parameters filter (1/2, 3/10).

Description

The function `ipprRemap` is declared in the `ippr.h` file. It operates with volume of interest (VOI).

This function transforms the source volume by remapping its voxels. Voxel remapping is performed using *pxMap*, *pyMap* and *pzMap* buffers to look-up the coordinates of the source volume voxel that is written to the target destination volume voxel. The application has to supply these look-up tables.

The remapping of the source voxels to the destination voxels is made according to the following formulas:

dst_voxel[*i*, *j*, *k*] = *src_voxel*[*pxMap*[*i*, *j*, *k*], *pyMap*[*i*, *j*, *k*], *pzMap* [*i* , *j* ,*k*]]

where *i*, *j* , *k* are the *x*-, *y*- and *z*-coordinates of the target destination volume voxel *dst_voxel*;

pxMap[*i*, *j*, *k*] contains the *x*-coordinates of the source volume voxels *src_voxel* that are written to *dst_voxel*.

pyMap[*i*, *j*, *k*] contains the *y*-coordinates of the source volume voxels *src_voxel* that are written to *dst_voxel*.

pzMap[*i*, *j*, *k*] contains the *z*-coordinates of the source volume voxels *src_voxel* that are written to *dst_voxel*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or a warning.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error condition if width, or height, or depth of the source and destination volumes has zero or negative value.
`ippStsInterpolationErr`	Indicates an error condition if *interpolation* has an illegal value.
`ippStsWrongIntersectVOI`	Indicates a warning if *srcVoi* has not intersection with the source volume, operation is not performed.

3D General Linear Filters

The functions described in this section perform filtering of 3D data.

FilterGetBufSize

Calculates the size of the working buffer.

Syntax

```
IppStatus ipprFilterGetBufSize(IpprVolume dstVolume, IpprVolume kernelVolume, int nChannel,
int* pBufferSize);
```

Parameters

dstVolume	Size of the processed volume.
kernelVolume	Size of the kernel volume.
nChannel	Number of channels or planes, possible value is one.
pBufferSize	Pointer to the size of the external buffer.

Description

The function `ipprFilterGetBufSize` is declared in the `ippr.h` file. It operates with VOI. This function computes the size of the working buffer *pBufferSize* that is required for the function ipprFilter.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error.
`ippStsNullPtrErr`	Indicates an error condition if *pBufferSize* pointer is `NULL`.
`ippStsNumChannelErr`	Indicates an error condition if *nChannel* has an illegal value.
`ippStsSizeErr`	Indicates an error condition if *dstVolume* or *kernelVolume* has a field with zero or negative value.

Filter

Filters a volume using a general cuboidal kernel.

Syntax

```
IppStatus ipprFilter_16s_C1PV(const Ipp16s* const pSrc[], int srcStep, Ipp16s* const
pDst[], int dstStep, IpprVolume dstVolume, const Ipp32s* pKernel, IpprVolume kernelVolume,
IpprPoint anchor, int divisor, Ipp8u* pBuffer);
```

Parameters

pSrc	Array of pointers to the planes in the source volume.
srcStep	Distance in bytes between starts of consecutive lines in each plane of the source volume.

pDst	Array of pointers to the planes in the destination volume.
dstStep	Distance in bytes between starts of consecutive lines in each plane of the destination volume.
dstVolume	Size of the processed volume.
pKernel	Pointers to the kernel values.
kernelVolume	Size of the kernel volume.
anchor	Anchor 3d-cell specifying the cuboidal kernel alignment with respect to the position of the input voxel.
divisor	The integer value by which the computed result is divided.
pBuffer	Pointer to the external buffer.

Description

The function `ipprFilter` is declared in the `ippr.h` file. It operates with VOI. This function uses the general cuboidal kernel of size *kernelVolume* to filter a volume VOI. This function sums the products between the kernel coefficients *pKernel* and voxel values taken over the source voxel neighborhood defined by *kernelVolume* and *anchor*. The anchor 3d-cell is specified by its coordinates *anchor.x*, *anchor.y* and *anchor.z* in the coordinate system associated with the right bottom back corner of the kernel. Note the kernel coefficients are used in inverse order. The sum is written to the destination voxel. To ensure valid operation when volume boundary voxels are processed, the application must correctly define additional border voxels.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error.
`ippStsNullPtrErr`	Indicates an error condition if *pSrc*, *pDst*, *pKernel* or *pBuffer* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error condition if *dstVolume* or *kernelVolume* has a field with zero or negative value.
`ippStsDivisorErr`	Indicates an error condition if the divisor value is zero.

Math Functions for 2D, 3D, and 4D Vectors 10

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) mathematical functions for two-dimensional (2D), three-dimensional (3D), and four-dimensional (4D) vectors.

The Intel IPP math functions for 2D, 3D, and 4D vectors perform the same mathematical operations as the math library contained in the D3DX 10 utility library of the Microsoft DirectX* Software Development Kit, but the Intel IPP functions use a different application programmer interface (API). Therefore, this chapter does not provide detailed descriptions of some functions. For detailed descriptions of such functions, refer to the documentation on the DirectX* SDK.

Math Functions for 2D, 3D, and 4D Vectors

Function Base Name	Operation
`Len`	Computes the length of a 2D, 3D, or 4D vector.
`LenSqr`	Computes the squared length of a 2D, 3D, or 4D vector.
`DotProduct`	Computes the dot product of two 2D, 3D, or 4D vectors.
`Add`	Adds two 2D, 3D, or 4D vectors.
`Sub`	Subtracts one 2D, 3D, or 4D vector from another.
`Min`	Computes the component-wise minimum of two 2D, 3D, or 4D vectors.
`Max`	Computes the component-wise maximum of two 2D, 3D, or 4D vectors.
`MulC`	Multiplies a 2D, 3D, or 4D vector by a constant.
`InterpolationLinear`	Performs linear interpolation of two 2D, 3D, or 4D vectors.
`Normalize`	Normalizes a 2D, 3D, or 4D vector.
`Hermite`	Performs the Hermite interpolation of two 2D, 3D, or 4D vectors.
`InterpolationCatmullRom`	Performs the Catmull-Rom interpolation of four 2D, 3D, or 4D vectors.
`BaryCentric`	Computes barycentric coordinates of a vector.
`CrossProduct`	Computes the cross product of two 2D, 3D, or 4D vectors.
`AffineTransform`	Performs an affine transformation of a 2D, 3D, or 4D vector or array of vectors in a Homogeneous coordinate space.

Function Base Name	Operation
AffineTransformCoord	Performs an affine transformation of coordinates of a 2D or 3D vector or each vector in an array in a Homogeneous coordinate space.
AffineTransformNormal	Performs an affine transformation of the normal of a 2D or 3D vector or each vector in an array in a Homogeneous coordinate space.
Project	Projects a 3D vector or array of vectors to the screen.
Unproject	Unprojects a 3D vector or array of vectors projected to the screen.
Invert	Inverts a 4x4 matrix.
Identity	Initializes the 4x4 identity matrix.
IsIdentity	Checks whether a 4x4 matrix is the identity matrix.
Det	Computes the determinant of a 4x4 matrix.
Transpose	Transposes a 4x4 matrix.
MulTranspose	Multiplies two 4x4 matrices and transposes the result.
Scaling	Computes the 4x4 scaling matrix.
Translation	Computes the 4x4 translation matrix.
RotationX, RotationY, RotationZ	Computes the 4x4 matrix of the rotation around the x, y, or z axis.
RotationAxis	Computes the 4x4 matrix of the rotation around an arbitrary axis.
RotationYawPitchRoll	Computes the 4x4 rotation matrix defined by the Tait-Bryan angles.
LookAtRH, LookAtLH	Computes the 4x4 right-handed or left-handed look-at matrix.
PerspectiveRH, PerspectiveLH	Computes the 4x4 matrix of the right-handed or left-handed perspective projection.
PerspectiveFovRH, PerspectiveFovLH	Computes the 4x4 matrix of the right-handed or left-handed perspective projection based on a field of view.
PerspectiveOffCenterRH, PerspectiveOffCenterLH	Computes the 4x4 matrix of right-handed or left-handed customized perspective projection.
OrthoRH, OrthoLH	Computes the 4x4 matrix of the right-handed or left-handed orthographic projection.
OrthoOffCenterRH, OrthoOffCenterLH	Computes the 4x4 matrix of the right-handed or left-handed customized orthographic projection.

Types of the Operands

The Intel IPP math functions for 2D, 3D, and 4D vectors operate on the following data objects:

- 2D vector,

  represented by a continuous array of two float numbers.
- 3D vector,

  represented by a continuous array of three float numbers.
- 4D vector,

  represented by a continuous array of four float numbers.
- 4x4 matrix,

  represented by a continuous array of 16 float numbers.

For all these objects, continuous arrays mean that *stride2*= sizeof(float) for the vectors and matrices, and *stride1*= 4*sizeof(float) for the matrices.

See Also

- Vector
- Matrix
- Strides

Naming Conventions

Names of the Intel IPP math functions for 2D, 3D, and 4D vectors have the same general format as the rest of Intel IPP functions for small matrices:

`ippm<name>_<objects>_<datatype>`.

However, the `<objects>` part is specific to the math functions for 2D, 3D, and 4D vectors.

`<objects> = <objecttype1>[<objecttype2>]`,

where `<objecttype1>` and `<objecttype2>` describe the type of the respective source object passed to the function and may be one of the following:

v2	2D vector
v2a	array of 2D vectors
v3	3D vector
v3a	array of 3D vectors
v4	4D vector
v4a	array of 4D vectors
m4	4x4 matrix

For example:

- The `ippmOrthoOffCenterRH_m3_32f` function accepts a 3D source vector.
- The `ippmAffineTransformNormal_m4v2a_32f` function accepts a 4x4 matrix and an array of 2D vectors as source parameters.

- The `ippmAffineTransform_m4v4_32f` function accepts a 4x4 matrix and a 4D vector as source parameters.

See Also

- Function Naming
- Objects

Len

Computes the length of a 2D, 3D, or 4D vector.

Syntax

```
IppStatus ippmLen_v2_32f (const Ipp32f* pSrc, Ipp32f* pDst);
IppStatus ippmLen_v3_32f (const Ipp32f* pSrc, Ipp32f* pDst);
IppStatus ippmLen_v4_32f (const Ipp32f* pSrc, Ipp32f* pDst);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the length of the vector.

Description

The functions `ippmLen_v2_32f, ippmLen_v3_32f`, and `ippmLen_v4_32f` are declared in the `ippm.h` file. These functions compute the length of a 2D, 3D, and 4D vector, respectively.

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

LenSqr

Computes the squared length of a 2D, 3D, or 4D vector.

Syntax

```
IppStatus ippmLenSqr_v2_32f (const Ipp32f* pSrc, Ipp32f* pDst);
IppStatus ippmLenSqr_v3_32f (const Ipp32f* pSrc, Ipp32f* pDst);
IppStatus ippmLenSqr_v4_32f (const Ipp32f* pSrc, Ipp32f* pDst);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the value of the squared length of the vector.

Description

The functions ippmLenSqr_v2_32f, ippmLenSqr_v3_32f, and ippmLenSqr_v4_32f are declared in the ippm.h file. These functions compute the squared length of a 2D, 3D, and 4D vector, respectively.

Return Values

ippStsNoErr	Indicates no parameter checking.

See Also

- Types of the Operands

DotProduct

Computes the dot product of two 2D, 3D, or 4D vectors.

Syntax

```
IppStatus ippmDotProduct_v2_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
IppStatus ippmDotProduct_v3_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
IppStatus ippmDotProduct_v4_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the value of dot product.

Description

The functions ippmDotProduct_v2_32f, ippmDotProduct_v3_32f, and ippmDotProduct_v4_32f are declared in the ippm.h file. These functions compute the dot product of 2D, 3D, and 4D vectors, respectively.

Return Values

ippStsNoErr	Indicates no parameter checking.

See Also

- Types of the Operands

Add

Adds two 2D, 3D, or 4D vectors.

Syntax

```
IppStatus ippmAdd_v2_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
IppStatus ippmAdd_v3_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
IppStatus ippmAdd_v4_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the sum of the source vectors.

Description

The functions `ippmAdd_v2_32f`, `ippmAdd_v3_32f`, and `ippmAdd_v4_32f` are declared in the `ippm.h` file. These functions add two 2D, 3D, and 4D vectors, respectively.

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

Sub

Subtracts one 2D, 3D, or 4D vector from another.

Syntax

```
IppStatus ippmSub_v2_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
IppStatus ippmSub_v3_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
IppStatus ippmSub_v4_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
```

Parameters

pSrc1	Pointer to the minuend source vector.
pSrc2	Pointer to the subtrahend source vector.
pDst	Pointer to the difference of the source vectors.

Description

The functions `ippmSub_v2_32f`, `ippmSub_v3_32f`, and `ippmSub_v4_32f` are declared in the `ippm.h` file. These functions subtract 2D, 3D, and 4D vectors, respectively.

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

Min

Computes the component-wise minimum of two 2D, 3D, or 4D vectors.

Syntax

```
IppStatus ippmMin_v2_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
IppStatus ippmMin_v3_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
IppStatus ippmMin_v4_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the resulting vector of the operation.

Description

The functions `ippmMin_v2_32f, ippmMin_v3_32f`, and `ippmMin_v4_32f` are declared in the `ippm.h` file. These functions operate on 2D, 3D, and 4D vectors, respectively. The functions return the vector whose components are the minimum of the respective components of the source vectors.

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

Max

Computes the component-wise maximum of two 2D, 3D, or 4D vectors.

Syntax

```
IppStatus ippmMax_v2_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
IppStatus ippmMax_v3_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
IppStatus ippmMax_v4_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the resulting vector of the operation.

Description

The functions `ippmMax_v2_32f`, `ippmMax_v3_32f`, and `ippmMax_v4_32f` are declared in the `ippm.h` file. These functions operate on 2D, 3D, and 4D vectors, respectively. The functions return the vector whose components are the maximum of the respective components of the source vectors.

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

MulC

Multiplies a 2D, 3D, or 4D vector by a constant.

Syntax

```
IppStatus ippmMulC_v2_32f (const Ipp32f* pSrc, Ipp32f scale, Ipp32f* pDst);
IppStatus ippmMulC_v3_32f (const Ipp32f* pSrc, Ipp32f scale, Ipp32f* pDst);
IppStatus ippmMulC_v4_32f (const Ipp32f* pSrc, Ipp32f scale, Ipp32f* pDst);
```

Parameters

pSrc	Pointer to the source vector.
scale	The scale factor.
pDst	Pointer to the result of multiplication.

Description

The functions `ippmMulC_v2_32f`, `ippmMulC_v3_32f`, and `ippmMulC_v4_32f` are declared in the `ippm.h` file. These functions scale a 2D, 3D, and 4D vector, respectively.

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

InterpolationLinear

Performs linear interpolation of two 2D, 3D, or 4D vectors.

Syntax

```
IppStatus ippmInterpolationLinear_v2_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f
scale, Ipp32f* pDst);
```

```
IppStatus ippmInterpolationLinear_v3_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f
scale, Ipp32f* pDst);
IppStatus ippmInterpolationLinear_v4_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f
scale, Ipp32f* pDst);
```

Parameters

pSrc1	Pointer to the first source vector *src1*.
pSrc2	Pointer to the second source vector *src2*.
scale	The scale factor.
pDst	Pointer to the resulting vector *dst* of the interpolation.

Description

The functions `ippmInterpolationLinear_v2_32f`, `ippmInterpolationLinear_v3_32f`, and `ippmInterpolationLinear_v4_32f` are declared in the `ippm.h` file. These functions interpolate 2D, 3D, and 4D vectors, respectively, according to the formula *dst = src1 + scale*(src2-src1)*.

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

Normalize

Normalizes a 2D, 3D, or 4D vector.

Syntax

```
IppStatus ippmNormalize_v2_32f (const Ipp32f* pSrc, Ipp32f* pDst);
IppStatus ippmNormalize_v3_32f (const Ipp32f* pSrc, Ipp32f* pDst);
IppStatus ippmNormalize_v4_32f (const Ipp32f* pSrc, Ipp32f* pDst);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the normalized vector.

Description

The functions `ippmNormalize_v2_32f`, `ippmNormalize_v3_32f`, and `ippmNormalize_v4_32f` are declared in the `ippm.h` file. These functions normalize a 2D, 3D, or 4D vector, respectively, by dividing its components by its length.

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

Hermite

Performs the Hermite interpolation of two 2D, 3D, or 4D vectors.

Syntax

```
IppStatus ippmHermite_v2_32f (const Ipp32f* pSrc1, const Ipp32f* pSrcT1, const Ipp32f*
pSrc2, const Ipp32f* pSrcT2, Ipp32f scale, Ipp32f* pDst);
```

```
IppStatus ippmHermite_v3_32f (const Ipp32f* pSrc1, const Ipp32f* pSrcT1, const Ipp32f*
pSrc2, const Ipp32f* pSrcT2, Ipp32f scale, Ipp32f* pDst);
```

```
IppStatus ippmHermite_v4_32f (const Ipp32f* pSrc1, const Ipp32f* pSrcT1, const Ipp32f*
pSrc2, const Ipp32f* pSrcT2, Ipp32f scale, Ipp32f* pDst);
```

Parameters

pSrc1	Pointer to the first position vector.
pSrcT1	Pointer to the first tangent vector.
pSrc2	Pointer to the second position vector.
pSrcT2	Pointer to the second tangent vector.
scale	The scale factor.
pDst	Pointer to the resulting vector of the interpolation.

Description

The functions `ippmHermite_v2_32f`, `ippmHermite_v3_32f`, and `ippmHermite_v4_32f` are declared in the `ippm.h` file. These functions perform the Hermite interpolation of 2D, 3D, and 4D vectors, respectively.

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

InterpolationCatmullRom

Performs the Catmull-Rom interpolation of four 2D, 3D, or 4D vectors.

Syntax

```
IppStatus ippmInterpolationCatmullRom_v2_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
const Ipp32f* pSrc3, const Ipp32f* pSrc4, Ipp32f s, Ipp32f* pDst);
```

```
IppStatus ippmInterpolationCatmullRom_v3_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
const Ipp32f* pSrc3, const Ipp32f* pSrc4, Ipp32f s, Ipp32f* pDst);
```

```
IppStatus ippmInterpolationCatmullRom_v4_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
const Ipp32f* pSrc3, const Ipp32f* pSrc4, Ipp32f s, Ipp32f* pDst);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pSrc3	Pointer to the third source vector.
pSrc4	Pointer to the fourth source vector.
s	The weighing factor.
pDst	Pointer to the resulting vector of the interpolation.

Description

The functions `ippmInterpolationCatmullRom_v2_32f`, `ippmInterpolationCatmullRom_v3_32f`, and `ippmInterpolationCatmullRom_v4_32f` are declared in the `ippm.h` file. These functions performs the Catmull-Rom interpolation of 2D, 3D, and 4D vectors, respectively.

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

BaryCentric

Computes barycentric coordinates of a vector.

Syntax

```
IppStatus ippmBaryCentric_v2_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, const Ipp32f*
pSrc3, Ipp32f f, Ipp32f g, Ipp32f* pDst);
```

```
IppStatus ippmBaryCentric_v3_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, const Ipp32f*
pSrc3, Ipp32f f, Ipp32f g, Ipp32f* pDst);
```

```
IppStatus ippmBaryCentric_v4_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, const Ipp32f*
pSrc3, Ipp32f f, Ipp32f g, Ipp32f* pDst);
```

Parameters

pSrc1	Pointer to the first source vector *src1*.
pSrc2	Pointer to the second source vector *src2*.
pSrc3	Pointer to the third source vector *src3*.
f, g	Weighing factors.
pDst	Pointer to the vector *dst* in barycentric coordinates.

Description

The functions `ippmBaryCentric_v2_32f`, `ippmBaryCentric_v3_32f`, and `ippmBaryCentric_v4_32f` are declared in the `ippm.h` file. These functions operate on 2D, 3D, and 4D vectors, respectively. The functions return the vector in barycentric coordinates computed according to the formula:

*dst = src1 + f * (src2 - src1) + g * (src3 - src1).*

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

CrossProduct

Computes the cross product of two 2D, 3D, or 4D vectors.

Syntax

```
IppStatus ippmCrossProduct_v2_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
IppStatus ippmCrossProduct_v3_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
IppStatus ippmCrossProduct_v4_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the resulting vector of the operation.

Description

The functions `ippmCrossProduct_v2_32f`, `ippmCrossProduct_v3_32f`, and `ippmCrossProduct_v4_32f` are declared in the `ippm.h` file. These functions compute the cross product of 2D, 3D, and 4D vectors, respectively.

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

AffineTransform

Performs an affine transformation of a 2D, 3D, or 4D vector or array of vectors in a Homogeneous coordinate space.

Syntax

IppStatus ippmAffineTransform_m4v2_32f (const Ipp32f* *pSrc1*, const Ipp32f* *pSrc2*, Ipp32f* *pDst*);

IppStatus ippmAffineTransform_m4v3_32f (const Ipp32f* *pSrc1*, const Ipp32f* *pSrc2*, Ipp32f* *pDst*);

IppStatus ippmAffineTransform_m4v4_32f (const Ipp32f* *pSrc1*, const Ipp32f* *pSrc2*, Ipp32f* *pDst*);

IppStatus ippmAffineTransform_m4v2a_32f (const Ipp32f* *pSrc1*, const Ipp32f* *pSrc2*, int src2*Stride0*, Ipp32f* *pDst*, int *dstStride0*, int *count*);

IppStatus ippmAffineTransform_m4v3a_32f (const Ipp32f* *pSrc1*, const Ipp32f* *pSrc2*, int src2*Stride0*, Ipp32f* *pDst*, int *dstStride0*, int *count*);

IppStatus ippmAffineTransform_m4v4a_32f (const Ipp32f* *pSrc1*, const Ipp32f* *pSrc2*, int src2*Stride0*, Ipp32f* *pDst*, int *dstStride0*, int *count*);

Parameters

pSrc1	Pointer to the source 4x4 matrix *src1*.
pSrc2	Pointer to the source vector *src2* or array of vectors.
src2Stride0	Stride between the vectors in the source array.
pDst	Pointer to the destination vector *dst* or array of vectors.
dstStride0	Stride between the vectors in the destination array.
count	The number of vectors in the arrays.

Description

The functions ippmAffineTransform_m4v2_32f, ippmAffineTransform_m4v3_32f, ippmAffineTransform_m4v4_32f, ippmAffineTransform_m4v2a_32f, ippmAffineTransform_m4v3a_32f, and ippmAffineTransform_m4v4a_32f are declared in the ippm.h file. These functions perform an affine transformation of a vector or array of vectors in a Homogeneous coordinate space. The transformation is determined by the *src1* matrix. The functions whose names contain the v2, v3, or v4 characters in the descriptor, operate on 2D, 3D, or 4D vectors, respectively. The functions whose names contain the a character before the underscore transform vector arrays, whereas the other functions transform vectors.

Provided that the 4x4 matrix *src1* is represented as a 16-element array, the transformation uses the following formulae, depending on the dimension of the vectors:

- For 2D vectors,

  *dst[0] = src1[0] * src2[0] + src1[4] * src2[1] + src1[12]*

  *dst[1] = src1[1] * src2[0] + src1[5] * src2[1] + src1[13]*

```
dst[2] = src1[2] * src2[0] + src1[6] * src2[1] + src1[14]
dst[3] = src1[3] * src2[0] + src1[7] * src2[1] + src1[15]
```

- For 3D vectors,

```
dst[0] = src1[0] * src2[0] + src1[4] * src2[1] + src1[ 8] * src2[2] + src1[12]
dst[1] = src1[1] * src2[0] + src1[5] * src2[1] + src1[ 9] * src2[2] + src1[13]
dst[2] = src1[2] * src2[0] + src1[6] * src2[1] + src1[10] * src2[2] + src1[14]
dst[3] = src1[3] * src2[0] + src1[7] * src2[1] + src1[11] * src2[2] + src1[15]
```

- For 4D vectors,

```
dst[0] = src1[0] * src2[0] + src1[4] * src2[1] + src1[ 8] * src2[2] + src1[12] * pSrc2[3]
dst[1] = src1[1] * src2[0] + src1[5] * src2[1] + src1[ 9] * src2[2] + src1[13] * pSrc2[3]
dst[2] = src1[2] * src2[0] + src1[6] * src2[1] + src1[10] * src2[2] + src1[14] * pSrc2[3]
dst[3] = src1[3] * src2[0] + src1[7] * src2[1] + src1[11] * src2[2] + src1[15] * pSrc2[3]
```

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

AffineTransformCoord

Performs an affine transformation of coordinates of a 2D or 3D vector or each vector in an array in a Homogeneous coordinate space.

Syntax

```
IppStatus ippmAffineTransformCoord_m4v2_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);

IppStatus ippmAffineTransformCoord_m4v3_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);

IppStatus ippmAffineTransformCoord_m4v2a_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, int src2Stride0, Ipp32f* pDst, int dstStride0, int count);

IppStatus ippmAffineTransformCoord_m4v3a_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, int src2Stride0, Ipp32f* pDst, int dstStride0, int count);
```

Parameters

pSrc1	Pointer to the source 4x4 matrix *src1*.
pSrc2	Pointer to the source vector *src2* or array of vectors.
src2Stride0	Stride between the vectors in the source array.

pDst	Pointer to the destination vector *dst* or array of vectors.
dstStride0	Stride between the vectors in the destination array.
count	The number of vectors in the arrays.

Description

The functions `ippmAffineTransformCoord_m4v2_32f`, `ippmAffineTransformCoord_m4v3_32f`, `ippmAffine-TransformCoord_m4v2a_32f`, and `ippmAffineTransformCoord_m4v3a_32f` are declared in the `ippm.h` file. These functions perform an affine transformation of coordinates of a single vector or each vector in an array in a Homogeneous coordinate space. The transformation is determined by the *src1* matrix. The functions whose names contain the v2 or v3 characters in the descriptor, operate on 2D or 3D vectors, respectively. The functions `ippmAffineTransformCoord_m4v2a_32f` and `ippmAffineTransformCoord_m4v3a_32f` operate on vector arrays, whereas the other functions operate on vectors.

Assume *length(src2)* is the length of the source vector. Provided that the 4x4 matrix *src1* is represented as a 16-element array, the transformation uses the following formulae, depending on the dimension of the vectors:

- For 2D vectors,

  *dst[0] = (src1[0] * src2[0] + src1[4] * src2[1]) / length(src2)*

  *dst[1] = (src1[1] * src2[0] + src1[5] * src2[1]) / length(src2)*

- For 3D vectors,

  *dst[0] = (src1[0] * src2[0] + src1[4] * src2[1] + src1[8] * src2[2]) / length(src2)*

  *dst[1] = (src1[1] * src2[0] + src1[5] * src2[1] + src1[9] * src2[2]) / length(src2)*

  *dst[2] = (src1[2] * src2[0] + src1[6] * src2[1] + src1[10] * src2[2]) / length(src2)*

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

AffineTransformNormal

Performs an affine transformation of the normal of a 2D or 3D vector or each vector in an array in a Homogeneous coordinate space.

Syntax

```
IppStatus ippmAffineTransformNormal_m4v2_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
Ipp32f* pDst);
```

```
IppStatus ippmAffineTransformNormal_m4v3_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
Ipp32f* pDst);
```

```
IppStatus ippmAffineTransformNormal_m4v2a_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
int src2Stride0, Ipp32f* pDst, int dstStride0, int count);
```

```
IppStatus ippmAffineTransformNormal_m4v3a_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
int src2Stride0, Ipp32f* pDst, int dstStride0, int count);
```

Parameters

pSrc1	Pointer to the source 4x4 matrix *src1*.
pSrc2	Pointer to the source vector *src2* or array of vectors.
src2Stride0	Stride between the vectors in the source array.
pDst	Pointer to the destination vector *dst* or array of vectors.
dstStride0	Stride between the vectors in the destination array.
count	The number of vectors in the arrays.

Description

The functions `ippmAffineTransformNormal_m4v2_32f`, `ippmAffineTransformNormal_m4v3_32f`, `ippmAffineTransformNormal_m4v2a_32f`, and `ippmAffineTransformNormal_m4v3a_32f` are declared in the `ippm.h` file. These functions perform an affine transformation of the normal of a single vector or each vector in an array in a Homogeneous coordinate space. The transformation is determined by the *src1* matrix. The functions whose names contain the v2 or v3 characters in the descriptor, operate on 2D or 3D vectors, respectively. The functions `ippmAffineTransformNormal_m4v2a_32f` and `ippmAffineTransformNormal_m4v3a_32f` operate on vector arrays, whereas the other functions operate on vectors.

Because the 4x4 matrix *src1* can be represented as a 16-element array, the transformation uses the following formulae, depending on the dimension of the vectors:

- For 2D vectors,

  *dst[0] = src1[0] * src2[0] + src1[4] * src2[1]*

  *dst[1] = src1[1] * src2[0] + src1[5] * src2[1]*

- For 3D vectors,

  *dst[0] = src1[0] * src2[0] + src1[4] * src2[1] + src1[8] * src2[2]*

  *dst[1] = src1[1] * src2[0] + src1[5] * src2[1] + src1[9] * src2[2]*

  *dst[2] = src1[2] * src2[0] + src1[6] * src2[1] + src1[10] * src2[2]*

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

Project

Projects a 3D vector or array of vectors to the screen.

Syntax

IppStatus ippmProject_m4v3_32f (const Ipp32f* *pWorld*, const Ipp32f* *pView*, const Ipp32f* *pProj*, const Ipp32f* *pSrc*, Ipp32f *vpWidth*, Ipp32f *vpHeight*, Ipp32f *vpZnear*, Ipp32f *vpZfar*, Ipp32f* *pDst*);

IppStatus ippmProject_m4v3a_32f (const Ipp32f* *pWorld*, const Ipp32f* *pView*, const Ipp32f* *pProj*, const Ipp32f* *pSrc*, int *srcStride0*, Ipp32f *vpWidth*, Ipp32f *vpHeight*, Ipp32f *vpZnear*, Ipp32f *vpZfar*, Ipp32f* *pDst*, int *dstStride0*, int *count*);

Parameters

pWorld	Pointer to the world matrix.
pView	Pointer to the viewport matrix.
pProj	Pointer to the projection matrix.
pSrc	Pointer to the source vector or array of vectors.
srcStride0	Stride between the vectors in the source array.
vpWidth	Width of the viewport.
vpHeight	Height of the viewport.
vpZnear	Minimum depth of the viewport.
vpZfar	Maximum depth of the viewport.
pDst	Pointer to the destination vector or array of vectors.
dstStride0	Stride between the vectors in the destination array.
count	The number of vectors in the arrays.

Description

The functions ippmProject_m4v3_32f and ippmProject_m4v3a_32f are declared in the ippm.h file. The ippmProject_m4v3_32f function projects a 3D vector to the screen. The ippmProject_m4v3a_32f function projects an array of 3D vectors to the screen.

Return Values

ippStsNoErr	Indicates no parameter checking.

See Also

- Types of the Operands

Unproject

Unprojects a 3D vector or array of vectors projected to the screen.

Syntax

`IppStatus ippmUnproject_m4v3_32f (const Ipp32f* pWorld, const Ipp32f* pView, const Ipp32f* pProj, const Ipp32f* pSrc, Ipp32f vpWidth, Ipp32f vpHeight, Ipp32f vpZnear, Ipp32f vpZfar, Ipp32f* pDst);`

`IppStatus ippmUnproject_m4v3a_32f (const Ipp32f* pWorld, const Ipp32f* pView, const Ipp32f* pProj, const Ipp32f* pSrc, int srcStride0, Ipp32f vpWidth, Ipp32f vpHeight, Ipp32f vpZnear, Ipp32f vpZfar, Ipp32f* pDst, int dstStride0, int count);`

Parameters

pWorld	Pointer to the world matrix.
pView	Pointer to the viewport matrix.
pProj	Pointer to the projection matrix.
pSrc	Pointer to the source vector or array of vectors.
srcStride0	Stride between the vectors in the source array.
vpWidth	Width of the viewport.
vpHeight	Height of the viewport.
vpZnear	Minimum depth of the viewport.
vpZfar	Maximum depth of the viewport.
pDst	Pointer to the destination vector or array of vectors.
dstStride0	Stride between the vectors in the destination array.
count	The number of vectors in the arrays.

Description

The functions `ippmUnproject_m4v3_32f` and `ippmUnproject_m4v3a_32f` are declared in the `ippm.h` file. These functions unproject a 3D vector and an array of 3D vectors, respectively. The `ippmUnproject_m4v3_32f` function performs an inverse operation of the `ippmProject_m4v3_32f` function. The `ippmUnproject_m4v3a_32f` function performs an inverse operation of the `ippmProject_m4v3a_32f` function.

Return Values

`ippStsNoErr`	Indicates no parameter checking.

See Also

- Types of the Operands

Invert

Inverts a 4x4 matrix.

Syntax

`IppStatus ippmInvert_m4_32f (const Ipp32f* pSrc, Ipp32f* pDstDet, Ipp32f* pDst);`

Parameters

pSrc	Pointer to the source 4x4 matrix.
pDstDet	Pointer to the determinant of the source matrix.
pDst	Pointer to the inverted matrix.

Description

The functions `ippmInvert_m4_32f` is declared in the `ippm.h` file. The function inverts the source 4x4 matrix.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsSingularErr`	Indicates an error when the source matrix is singular.

See Also

- Types of the Operands

Identity

Initializes the 4x4 identity matrix.

Syntax

`IppStatus ippmIdentity_m4_32f (Ipp32f* pDst);`

Parameters

pDst	Pointer to the identity matrix.

Description

The functions `ippmIdentity_m4_32f` is declared in the `ippm.h` file. The function loads the 4x4 identity matrix into memory and stores the result in the destination object.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

IsIdentity

Checks whether a 4x4 matrix is the identity matrix.

Syntax

```
IppStatus ippmIsIdentity_m4_32f (const Ipp32f* pSrc, Ipp32s* pDst);
```

Parameters

pSrc	Pointer to the source 4x4 matrix.
pDst	Pointer to the result of checking.

Description

The functions `ippmIsIdentity_m4_32f` is declared in the `ippm.h` file. The function checks whether the source matrix is the identity matrix and returns:

- 1 if the matrix is the identity matrix
- 0 otherwise.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

Det

Computes the determinant of a 4x4 matrix.

Syntax

```
IppStatus ippmDet_m4_32f (const Ipp32f* pSrc, Ipp32f* pDst);
```

Parameters

pSrc	Pointer to the source 4x4 matrix.
pDst	Pointer to the determinant of the matrix.

Description

The functions `ippmDet_m4_32f` is declared in the `ippm.h` file. The function computes the determinant of the source matrix.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

Transpose

Transposes a 4x4 matrix.

Syntax

```
IppStatus ippmTranspose_m4_32f (const Ipp32f* pSrc, Ipp32f* pDst);
```

Parameters

pSrc	Pointer to the source 4x4 matrix.
pDst	Pointer to the transposed matrix.

Description

The functions `ippmTranspose_m4_32f` is declared in the `ippm.h` file. The function transposes the source matrix.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

MulTranspose

Multiplies two 4x4 matrices and transposes the result.

Syntax

```
IppStatus ippmMulTranspose_m4_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst);
```

Parameters

pSrc1	Pointer to the first source matrix.
pSrc2	Pointer to the second source matrix.
pDst	Pointer to the transposed result of the matrix multiplication.

Description

The function `ippmMulTranspose_m4_32f` is declared in the `ippm.h` file. This function multiplies the source 4x4 matrices and transposes the resulting matrix.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

Scaling

Computes the 4x4 scaling matrix.

Syntax

```
IppStatus ippmScaling_m4_32f (Ipp32f sx, Ipp32f sy, Ipp32f sz, Ipp32f* pDst);
```

Parameters

px, py, pz	Scale factors for the x, y, and z axes, respectively.
pDst	Pointer to the scaling matrix *dst*.

Description

The function `ippmScaling_m4_32f` is declared in the `ippm.h` file. This function computes the 4x4 scaling matrix for the given scale factors, as follows:

```
        |sx  0  0  0 |
   dst = | 0 sy  0  0 |
        | 0  0 sz  0 |
        | 0  0  0  1 |
```

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

Translation

Computes the 4x4 translation matrix.

Syntax

```
IppStatus ippmTranslation_m4_32f (Ipp32f sx, Ipp32f sy, Ipp32f sz, Ipp32f* pDst);
```

Parameters

px, py, pz	x, y, and z components of the translation, respectively.
pDst	Pointer to the translation matrix *dst*.

Description

The function `ippmTranslation_m4_32f` is declared in the `ippm.h` file. This function computes the 4x4 translation matrix for the given translation components, as follows:

```
        | 1  0  0  sx |
  dst = | 0  1  0  sy |
        | 0  0  1  sz |
        | 0  0  0  1  |
```

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

RotationX, RotationY, RotationZ

Computes the 4x4 matrix of the rotation around the x, y, or z axis.

Syntax

```
IppStatus ippmRotationX_m4_32f (Ipp32f angle, Ipp32f* pDst);
IppStatus ippmRotationY_m4_32f (Ipp32f angle, Ipp32f* pDst);
IppStatus ippmRotationZ_m4_32f (Ipp32f angle, Ipp32f* pDst);
```

Parameters

angle	Rotation angle.
pDst	Pointer to the rotation matrix *dst*.

Description

The functions `ippmRotationX_m4_32f`, `ippmRotationY_m4_32f`, and `ippmRotationZ_m4_32f` are declared in the `ippm.h` file. These functions compute the 4x4 rotation matrix for the given angle, as follows:

- For the x axis,

```
      | 1       0           0       0 |
dst = | 0   cos(angle)  sin(angle)  0 |
      | 0  -sin(angle)  cos(angle)  0 |
      | 0       0           0       1 |
```

- For the y axis,

```
      | cos(angle)  0  -sin(angle)  0 |
dst = |     0       1       0       0 |
      | sin(angle)  0   cos(angle)  0 |
      |     0       0       0       1 |
```

- For the z axis,

```
         |  cos(angle)  sin(angle)  0  0 |
   dst = | -sin(angle)  cos(angle)  0  0 |
         |      0           0       1  0 |
         |      0           0       0  1 |
```

The `ippmRotationX_m4_32f`, `ippmRotationY_m4_32f`, and `ippmRotationZ_m4_32f` functions compute the matrix for the rotation around the x, y, and z axes, respectively.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

RotationAxis

Computes the 4x4 matrix of the rotation around an arbitrary axis.

Syntax

`IppStatus ippmRotationAxis_v3m4_32f (const Ipp32f* pSrc, Ipp32f angle, Ipp32f* pDst);`

Parameters

pSrc	Pointer to the rotation axis.
angle	Rotation angle.
pDst	Pointer to the rotation matrix.

Description

The function `ippmRotationAxis_v3m4_32f` is declared in the `ippm.h` file. This function computes the 4x4 matrix of the rotation around the given axis with the given angle.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

RotationYawPitchRoll

Computes the 4x4 rotation matrix defined by the Tait-Bryan angles.

Syntax

```
IppStatus ippmRotationYawPitchRoll_m4_32f (Ipp32f yaw, Ipp32f pitch, Ipp32f roll, Ipp32f*
pDst);
```

Parameters

yaw	Rotation around the y axis.
pitch	Rotation around the x axis.
roll	Rotation around the z axis.
pDst	Pointer to the rotation matrix.

Description

The function `ippmRotationYawPitchRoll_m4_32f` is declared in the `ippm.h` file. This function computes the 4x4 rotation matrix defined by the given yaw, pitch, and roll.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

LookAtLH, LookAtRH

Computes the 4x4 right-handed or left-handed look-at matrix.

Syntax

```
IppStatus ippmLookAtRH_m4_32f (const Ipp32f* pSrcEye, const Ipp32f* pSrcAt, const Ipp32f*
pSrcUp, Ipp32f* pDst);
```

```
IppStatus ippmLookAtLH_m4_32f (const Ipp32f* pSrcEye, const Ipp32f* pSrcAt, const Ipp32f*
pSrcUp, Ipp32f* pDst);
```

Parameters

pSrcEye	Pointer to the eye.
pSrcAt	Pointer to the camera.
pSrcUp	Pointer to the current world's up.
pDst	Pointer to the look-at matrix.

Description

The functions `ippmLookAtRH_m4_32f` and `ippmLookAtLH_m4_32f` are declared in the `ippm.h` file. The functions compute the 4x4 look-at matrix defined by the parameters. The `ippmLookAtRH_m4_32f` function computes the right-handed look-at matrix, and `ippmLookAtLH_m4_32f` computes the left-handed look-at matrix.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

PerspectiveRH, PerspectiveLH

Computes the 4x4 matrix of the right-handed or left-handed perspective projection.

Syntax

```
IppStatus ippmPerspectiveRH_m4_32f (Ipp32f w, Ipp32f h, Ipp32f zNear, Ipp32f zFar, Ipp32f* pDst);

IppStatus ippmPerspectiveLH_m4_32f (Ipp32f w, Ipp32f h, Ipp32f zNear, Ipp32f zFar, Ipp32f* pDst);
```

Parameters

w	Width of the view at the near plane.
h	Height of the view at the near plane.
zNear	z value of the near plane.
zFar	z value of the far plane.
pDst	Pointer to the perspective projection matrix.

Description

The functions `ippmPerspectiveRH_m4_32f` and `ippmPerspectiveLH_m4_32f` are declared in the `ippm.h` file. The functions compute the 4x4 matrix of the perspective projection defined by the parameters. The `ippmPerspectiveRH_m4_32f` function computes the matrix of the right-handed projection, and `ippmPerspectiveLH_m4_32f` computes the matrix of the left-handed projection.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

PerspectiveFovRH, PerspectiveFovLH

Computes the 4x4 matrix of the right-handed or left-handed perspective projection based on a field of view.

Syntax

```
IppStatus ippmPerspectiveFovRH_m4_32f (Ipp32f fovy, Ipp32f aspect, Ipp32f zNear, Ipp32f
zFar, Ipp32f* pDst);
```

```
IppStatus ippmPerspectiveFovLH_m4_32f (Ipp32f fovy, Ipp32f aspect, Ipp32f zNear, Ipp32f
zFar, Ipp32f* pDst);
```

Parameters

fovy	Field of view in the y direction.
aspect	Aspect ratio.
zNear	z coordinate of the near plane.
zFar	z coordinate of the far plane.
pDst	Pointer to the perspective projection matrix.

Description

The functions `ippmPerspectiveFovRH_m4_32f` and `ippmPerspectiveFovLH_m4_32f` are declared in the `ippm.h` file. The functions compute the 4x4 matrix of the perspective projection defined by the parameters of the field of view. The `ippmPerspectiveFovRH_m4_32f` function computes the matrix of the right-handed projection, and `ippmPerspectiveFovLH_m4_32f` computes the matrix of the left-handed projection.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

PerspectiveOffCenterRH, PerspectiveOffCenterLH

Computes the 4x4 matrix of right-handed or left-handed customized perspective projection.

Syntax

```
IppStatus ippmPerspectiveOffCenterRH_m4_32f (Ipp32f left, Ipp32f right, Ipp32f bottom,
Ipp32f top, Ipp32f zNear, Ipp32f zFar, Ipp32f* pDst);
```

```
IppStatus ippmPerspectiveOffCenterLH_m4_32f (Ipp32f left, Ipp32f right, Ipp32f bottom,
Ipp32f top, Ipp32f zNear, Ipp32f zFar, Ipp32f* pDst);
```

Parameters

left	The minimum x value of the view.
right	The maximum x value of the view.
bottom	The minimum y value of the view.
top	The maximum y value of the view.
zNear	The minimum z value of the view.
zFar	The maximum z value of the view.
pDst	Pointer to the perspective projection matrix.

Description

The functions `ippmPerspectiveOffCenterRH_m4_32f` and `ippmPerspectiveOffCenterLH_m4_32f` are declared in the `ippm.h` file. The functions compute the 4x4 matrix of the perspective projection defined by the parameters. The `ippmPerspectiveOffCenterRH_m4_32f` function computes the matrix of the right-handed projection, and `ippmPerspectiveOffCenterLH_m4_32f` computes the matrix of the left-handed projection.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

OrthoRH, OrthoLH

Computes the 4x4 matrix of the right-handed or left-handed orthographic projection.

Syntax

```
IppStatus ippmOrthoRH_m4_32f (Ipp32f w, Ipp32f h, Ipp32f zNear, Ipp32f zFar, Ipp32f* pDst);
IppStatus ippmOrthoLH_m4_32f (Ipp32f w, Ipp32f h, Ipp32f zNear, Ipp32f zFar, Ipp32f* pDst);
```

Parameters

w	Width of the view.
h	Height of the view.
zNear	z value of the near plane.
zFar	z value of the far plane.
pDst	Pointer to the orthographic projection matrix.

Description

The functions `ippmOrthoRH_m4_32f` and `ippmOrthoLH_m4_32f` are declared in the `ippm.h` file. The functions compute the 4x4 matrix of the orthographic projection defined by the parameters. The `ippmOrthoRH_m4_32f` function computes the matrix of the right-handed projection, and `ippmOrthoLH_m4_32f` computes the matrix of the left-handed projection.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

OrthoOffCenterRH, OrthoOffCenterLH

Computes the 4x4 matrix of the right-handed or left-handed customized orthographic projection.

Syntax

```
IppStatus ippmOrthoOffCenterRH_m4_32f (Ipp32f left, Ipp32f right, Ipp32f bottom, Ipp32f
top, Ipp32f zNear, Ipp32f zFar, Ipp32f* pDst);
```

```
IppStatus ippmOrthoOffCenterLH_m4_32f (Ipp32f left, Ipp32f right, Ipp32f bottom, Ipp32f
top, Ipp32f zNear, Ipp32f zFar, Ipp32f* pDst);
```

Parameters

left	The minimum x value of the view.
right	The maximum x value of the view.
bottom	The minimum y value of the view.
top	The maximum y value of the view.
zNear	The minimum z value of the view.
zFar	The maximum z value of the view.
pDst	Pointer to the orthographic projection matrix.

Description

The functions `ippmOrthoOffCenterRH_m4_32f` and `ippmOrthoOffCenterLH_m4_32f` are declared in the `ippm.h` file. The functions compute the 4x4 matrix of the orthographic projection defined by the parameters. The `ippmOrthoOffCenterRH_m4_32f` function computes the matrix of the right-handed projection, and `ippmOrthoOffCenterLH_m4_32f` computes the matrix of the left-handed projection.

Return Values

`ippStsNoErr`	Indicates no error.

See Also

- Types of the Operands

Index

I

K

L

M

N

O

P

Q

R

S

T

U

V

W