Integrated Performance Primitives for Intel® Architecture

Reference Manual

Volume 1:
Signal Processing

Issued in U.S.A.
Document Number: A24968-2002

World Wide Web: http://developer.intel.com

Version	Version Information	Date
-1001	Original issue.	09/00
-1002	Documents IPP 1.0 final release. Functions NormDiff, AutoCorr, and ZeroMean have been added. Derivatives Functions section have been revised.	02/01
-1101	Documents IPP 1.1 beta release.	04/01
-2001	Documents IPP 2.0 beta release. General audio coding, MP3, and transcendental vector functions have been added. Speech recognition API have been revised.	08/01
-2002	Documents IPP 2.0 Gold release. New IPP common functions have been added. The set of arithmetic, vector initialization, statistical, and filtering functions have been expanded.	11/01

Contents

Chapter 9 Audio Coding Functions

Chapter 10 MP3 Audio Decoder

Overview

1

This manual describes the structure, operation and functions of the Integrated Performance Primitives (IPP) for Intel® architecture that operate on one-dimensional signals. This is the first volume of the IPP Reference Manual, which also comprises descriptions of the IPP for image and video processing (volume 2) and operations on small matrices (volume 3). The IPP software package supports many functions whose performance can be significantly enhanced on the Intel® architecture (IA), particularly using the MMX™ technology and Streaming SIMD Extensions.

The IPP for signal processing software is a collection of low-overhead, high-performance operations performed on one-dimensional (1D) data arrays.

This manual explains the IPP concepts as well as specific data type definitions and operation models used in the signal processing domain and provides detailed descriptions of the IPP signal processing functions.

This chapter introduces the IPP software and explains the organization of this manual.

About This Software

The IPP software enables taking advantage of the parallelism of the single-instruction, multiple-data (SIMD) instructions that comprise the core of the MMX technology and Streaming SIMD Extensions. These technologies improve the performance of computation-intensive signal and image processing applications.

Hardware and Software Requirements

The IPP software runs on personal computers that are based on Intel architecture processors and running Microsoft* Windows* 95, Windows 98, Windows 2000, or Windows NT*. The IPP integrates into the customer's application or library written in C or C++.

Platforms Supported

The IPP software runs on Windows platforms. The code and syntax used for function and variable declarations in this manual are written in the ANSI C style. However, versions of the IPP for different processors or operating systems may, of necessity, vary slightly.

About This Manual

This manual provides a background for the signal processing concepts used in the IPP software as well as detailed descriptions of the IPP functions for signal processing. The IPP functions are combined in groups by their functionality. Each group of functions is described in a separate chapter (chapters 3 through 11).

Manual Organization

This manual contains the following chapters:

Chapter 1	"Overview." Introduces the IPP software, provides information on manual organization, and explains notational conventions.
Chapter 2	"Integrated Performance Primitives Concepts." Specifies the supported data formats and operation modes.
Chapter 3	"Support Functions." Describes functions used to manipulate the signals, such as `Copy` and `Set`. Also describes data conversion and memory allocation functions.
Chapter 4	"Vector Initialization Functions." Describes initialization and sample-generating functions.
Chapter 5	"Essential Vector Functions." Describes vector manipulation functions.

Chapter 6	"Filtering Functions." Detais filtering operations that use linear and non-linear filters.
Chapter 7	"Transform Functions." Describes domain transform functions (Fourier, Wavelet, Cosine).
Chapter 8	"Speech Recognition Functions." Describes functions used in the speech recognition applications.
Chapter 9	"Audio Coding Functions." Includes general purpose functions applicable in several codecs and a number of specific functions for MPEG-4 audio codec.
Chapter 10	"MP3 Audio Decoder." Describes functions used to construct audio decoders compliant with the layer III portions of ISO/IEC MPEG-1 and MPEG-2 audio specifications.
Chapter 11	"Fixed-Accuracy Arithmetic Functions." Describes IPP fixed-accuracy vector mathematical functions suitable for multimedia and signal processing in real time applications.

The manual also includes a Glossary of terms, a Bibliography, and an Index.

Function Descriptions

In Chapters 3 through 11, each function is introduced by its short name (without the `ipps` prefix and modifiers) and a brief description of its purpose. This is followed by the function call sequence, definition of its arguments, and more detailed explanation of the function's purpose. The following sections are included in function description:

Arguments	Specifies all the function arguments.
Discussion	Defines the function and describes the operation performed by the function. This section may also include the code examples and descriptive equations.
Application Notes	If present, describes any special information which application programmers or other users of the function need to know.
Return Value	Explains the value returned by the function. Most commonly, it lists error codes that the function returns.

See Also	If present, lists the names of functions which perform related tasks.

Audience for This Manual

The manual is intended for the developers of signal processing applications and libraries. The audience is expected to be experienced in using C and to have a working knowledge of the vocabulary and principles of signal processing.

Online Version

This manual is available in an electronic format (Portable Document Format, or PDF). To obtain a hard copy of the manual, print the file using the printing capability of Adobe* Acrobat*, the tool used for the online presentation of the document.

Related Publications

For more information about signal processing concepts and algorithms, refer to the books and papers listed in the Bibliography.

Notational Conventions

In this manual, notational conventions include:

- Fonts used for distinction between the text and the code;
- Signal name conventions;
- Naming conventions.

Font Conventions

The following font conventions are used throughout the manual:

`THIS TYPE STYLE`	Used in the text for IPP constant identifiers; for example, `IPP_MAX_64S`.

`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, `void ippsFree()`.
`This type style`	Variables in arguments discussion; for example, *`val`*, *`srcLen`*.

Signal Name Conventions

In this manual, vectors and arrays are commonly used to represent a discrete 1D signal. The notation *x(n)* refers to a conceptual signal, while the notation *x*[*n*] refers to an actual vector. Both of these are annotated to indicate a specific finite range of values:

x[*n*], $0 \le n < len$.

Typically, the number of elements in vectors is denoted by *len*. Vector names contain square brackets as distinct from vector elements with current index *n*.

For example, the expression *pDst*[*n*] = *pSrc*[*n*] + *val* implies that each element *pDst*[*n*] of the vector *pDst* is computed for each *n* in the range from 0 to *len*-1. Special cases are regarded and described separately.

Naming Conventions

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

- Constant identifiers are in uppercase; for example, `IPP_MIN_64S`.
- All structures and enumerators, specific for the signal processing domain have the `Ipps` prefix, while those common for entire IPP software have the `Ipp` prefix; for example, `IppsROI`, `IppLibraryVersion`.

- All names of the functions used for signal processing have the `ipps` prefix. In code examples, you can distinguish the IPP interface functions from the application functions by this prefix.

NOTE. *In this manual, the* `ipps` *prefix in function names is always used in the code examples. In the text, this prefix is usually omitted when referring to the function group.*

- Each new part of a function name starts with an uppercase character, without underscore; for example, `ippsAddAllRowSum`.

For the detailed description of function names structure in IPP, see “Function Naming” on page 2-2.

Intel® Integrated Performance Primitives Concepts

2

This chapter explains the purpose and structure of the IPP software and discusses differences and connections with other members of the Intel® Performance Library Suite. This chapter also looks over some of the basic concepts used in the signal processing part of IPP, describes the supported data formats and operation modes, and defines function naming conventions in the manual.

Relation Between IPP and Other Libraries

The Intel Integrated Performance Primitives, like other members of the Intel Performance Library Suite, is a collection of high-performance code that performs domain-specific operations. It differs from current library offerings by providing a low-level, stateless interface.

Based on experience in developing and using other libraries of the Intel Performance Library Suite, IPP has the following major distinctive features:

- The IPP provides basic functions for creating applications in several different domains, such as signal processing, image and video processing, and operations on small matrices;
- IPP functions follow the same interface conventions including uniform naming rules and similar composition of prototypes for primitives that refer to different application domains;
- IPP functions use abstraction level which is best suited to achieve superior performance figures by the application programs.

To speed up performance, IPP functions are optimized to use all benefits of Intel architecture processors. Besides that, IPP functions do not use complicated data structures present in other libraries, which helps reduce overall execution overhead.

IPP is well-suited for cross-platform applications. For example, the functions developed for IA-32 platform can be readily ported to Itanium™-based platforms and StrongARM* platforms. In addition, each IPP function has its reference code written in ANSI C, which clearly presents the algorithm used and provides for compatibility with different operating systems.

Function Naming

Naming conventions for the IPP functions are similar for all covered domains. You can distinguish signal processing functions by the `ipps` prefix, while image and video processing functions have `ippi` prefix, and functions that are specific for operations on small matrices have `ippm` prefix in their names.

Function names in IPP have the following general format:

`ipp<domain><name>_<datatype>[_<descriptor>](<arguments>);`

The elements of this format are explained in the sections that follow.

Domain

The *domain* is a single character that expresses the subset of functionality to which a given function belongs. The current version of IPP supports the following domains:

`s`	for signals;
`i`	for images and video;
`m`	for matrices.

For example, function names that begin with `ipps` signify that respective functions are used for signal processing.

Name

The *name* is an abbreviation for the core operation that the function really does, for example `Set`, `Copy`, followed in some cases by a function-specific modifier:

`<name> = <operation>[_modifier]`

This modifier, if present, denotes a slight modification or variation of the given function. For example, the modifier `CToC` in the function `ippsFFTInv_CToC_32fc` signifies that the inverse fast Fourier transform operates on complex data.

Data Types

The *datatype* field indicates data types used by the function, in the following format:

`<1|8|16|32|64><u|s|f>[c]`

where an integer indicates the bit depth, *u* indicates "unsigned integer", *s* indicates "signed integer", *f* indicates "floating point", and *c* indicates "complex".

The current version of IPP supports the data types of the source and destination for signal processing functions listed in Table 2-1.

NOTE. *In the lists of function arguments, the* `Ipp` *prefix is written in the data type. For example, the 8-bit signed data is denoted as* `Ipp8s` *type.*

Table 2-1 Data Types Supported by IPP for Signal Processing

Bit Depth	Usual Type	IPP Type
8	`unsigned char`	`Ipp8u`
8	`signed char`	`Ipp8s`
16	`unsigned short`	`Ipp16u`
16	`signed short`	`Ipp16s`
16	`complex short`	`Ipp16sc`
32	`unsigned int`	`Ipp32u`
32	`signed int`	`Ipp32s`
32	`float`	`Ipp32f`
32	`complex float`	`Ipp32fc`
64	`__int64` (Windows) or `long long` (Linux)	`Ipp64s`
64	`double`	`Ipp64f`
64	`complex double`	`Ipp64fc`

For functions that operate on a single data type, the *datatype* field contains only one of the values listed above.

If a function operates on source and destination signals that have different data types, the respective data type identifiers are listed in the function name in order of source and destination as follows:

<*datatype*> = <*src1Depth*>[*src2Depth*][*dstDepth*]

For example, the function ippsDotProd_16s16sc_Sfs computes the dot product of 16-bit short and 16-bit complex short source vectors and stores the result in a 16-bit complex short destination vector. The *dstDepth* modifier is not present in the name because the second operand and the result are of the same type. The result is scaled and saturated.

Descriptor

The *descriptor* field further describes the data associated with the operation. It may contain implied parameters and/or indicate additional required parameters. To minimize the number of code branches in the function and thus reduce potentially unnecessary execution overhead, most of the general functions were split into separate primitive functions, with some of their parameters entering the primitive function name as descriptors.

However, some functions may still have parameters that determine internal operations (for example, ippsThreshold).

The following descriptors are used in signal processing functions:

I	Operation is in-place; not-in-place by default.
Sfs	Saturation and fixed scaling mode; no scaling by default.

The abbreviations of descriptors in function names are always presented in alphabetical order.

Not all functions have every abbreviation listed above. For example, in-place mode makes no sense for a copy operation.

Arguments

The *arguments* field specifies the function arguments.

The order of arguments is as follows:

1. All source operands. Constants follow vectors.
2. All destination operands. Constants follow vectors.
3. Other, operation-specific arguments.

The argument name have the following conventions:

- All arguments defined as pointers start with *p*, for example, *pPhase*, *pSrc*, *pSeed*; and arguments defined as values start with a lowercase letter, for example, *val*, *src*, *srcLen*.
- Each new part of a argument name starts with an uppercase character, without underscore; for example, *pSrc*, *lenSrc*, *pDlyLine*.
- Each argument name specifies its functionality. Source arguments are named *pSrc* or *src*, sometimes followed by names or numbers, e.g., *pSrc2*, *srcLen*. Output arguments are named *pDst* or *dst* followed by names or numbers, e.g., *pDst*, *dstLen*. For in-place operations, the input/ output argument contains the name *pSrcDst*.

Structures and Enumerators

This section describes the structures and enumerators used by the IPP for signal processing.

Library Version Structure

The `IppLibraryVersion` structure describes the current IPP software version. The main fields of this structure are:

- integer fields *major* and *minor*, which store version numbers;
- string field *Name* storing the IPP version name, e.g., “ippsa6”;
- string field *Version*, storing the version description, e.g., “v0.0 Alpha 0.0.3.3”.

Complex Data Structures

Complex numbers in IPP are described by the structures, containing two numbers of the respective data type which are real and imaginary parts of the complex number. For example, the single precision complex number is described by the `Ipp32fc` structure as follows:

```
typedef struct {
    Ipp32f re;
    Ipp32f im;
} Ipp32fc;
```

The following complex data types are defined: `Ipp16sc`, `Ipp32fc`, `Ipp64fc`.

Function Context Structures

Some IPP functions that perform such operations as filtering, Fourier and wavelet transforms, use context structures to store function-specific information.

For example, the `IppsFFTSpec` structure stores twiddle factors and bit reverse indexes needed in the fast Fourier transform.

These context-related structures are not defined in public headers, and you can not access the structure fields. It was done because the function context interpretation is processor dependent. Thus, you may only use context-related functions and may not create a function context as an automatic variable.

Enumerators

The `IppStatus` constant enumerates the status values returned by the IPP functions, indicating whether the operation was error-free or not. See “Error Reporting” on page 2-9 for more information on the set of valid status values and corresponding error messages for signal processing functions.

The `IppCmpOp` enumeration defines the type of relational operator to be used in threshold functions on page 5-62:

```
typedef enum {
    ippCmpLess,
    ippCmpLessEq,
    ippCmpEq,
```

```
        ippCmpGreaterEq,
        ippCmpGreater
    } IppCmpOp;
```

The `IppCnvrtRnd` enumeration defines the rounding mode to be used in conversion functions on page 5-49:

```
typedef enum {
    ippRndZero,
    ippRndNear
} IppCnvrtRnd;
```

The `IppHintAlgorithm` enumeration defines the type of code to be used in some transform operations, i. e. faster but less accurate, or vice-versa, more accurate but slower. For more information on using this enumerator, see Flag and Hint Arguments.

```
typedef enum {
    ippAlgHintNone,
    ippAlgHintFast,
    ippAlgHintAccurate
} IppHintAlgorithm;
```

The `IppCpuType` enumerates processor types returned by the `ippGetCpuType` function, see page 3-3.

```
typedef enum {
    ippCpuUnknown = 0x0,
    ippCpuPP,
    ippCpuPMX,
    ippCpuPPR,
    ippCpuPII,
    ippCpuPIII,
    ippCpuP4,
    ippCpuITP = 0x10
} IppCpuType;
```

Data Alignment

The IPP is built using the compiler option `/Zp8` which aligns the structure fields on the field's size or 8 bytes, if the size is greater than 8.

The IPP allows also to align the allocated memory pointer on 32 bytes by means of using the function `ippsMalloc`.

Integer Scaling

Some signal processing functions operating on integer data use scaling of the internally computed output results by the integer *scaleFactor*, which is specified as one of the function arguments. These functions have the `Sfs` descriptor in their names.

The scale factor can be negative, positive, or zero. Scaling is applied because internal computations are generally performed with a higher precision than the data types used for input and output signals.

NOTE. *The result of integer operations is always saturated to the destination data type range even when scaling is used.*

Scaling of an integer result is done by multiplying the output vector values by $2^{-scaleFactor}$ before the function returns. This helps retain either the output data range or its precision. Usually the scaling with a positive factor is performed by the shift operation. The result is rounded off to the nearest integer number.

For example, the integer `Ipp16s` result of the square operation `ippsSqr` for the input value 200 is equal to 32767 instead of 40000, that is, the result is saturated and the exact value can not be restored.

The scaling of the output value with the factor *scaleFactor* = 1 yields the result 20000 which is not saturated, and the exact value can be restored as 20000*2. Thus, the output data range is retained.

The following example shows how the precision can be partially retained by means of scaling.

The integer square root operation `ippsSqrt` (without scaling) for the input value 2 gives the result equal to 1 instead of 1.414. Scaling of the internally computed output value with the factor *scaleFactor* = -3 gives the result 11, and permits to restore the more precise value as $11*2^{-3} = 1.375$.

Error Reporting

The 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.

Table 2-2 lists status codes and corresponding messages reported by the IPP for signal processing.

Table 2-2 Error Status Values and Messages

Status	Value	Message
`ippStsToneMagnErr`	-38	Incorrect magnitude for tone generation
`ippStsToneFreqErr`	-37	Incorrect frequency for tone generating
`ippStsTonePhaseErr`	-36	Incorrect phase for tone generation
`ippStsTrnglMagnErr`	-35	Incorrect magnitude for triangle generation
`ippStsTrnglFreqErr`	-34	Incorrect frequency for triangle generation
`ippStsTrnglPhaseErr`	-33	Incorrect phase for triangle generation
`ippStsTrnglAsymErr`	-32	Incorrect asymmetry for triangle generation
`ippStsHugeWinErr`	-31	Incorrect size of the Kaiser window
`ippStsJaehneErr`	-30	Incorrect value of the magnitude
`ippStsStepErr`	-29	Step value is less or equal zero
`ippStsDlyLineIndexErr`	-28	Incorrect value of the delay line sample index
`ippStsStrideErr`	-27	Stride value is less than the row length
`ippStsEpsValErr`	-26	Negative epsilon value error
`ippStsScaleRangeErr`	-25	Scale bounds are out of range
`ippStsThresholdErr`	-24	Incorrect threshold bounds
`ippStsWtOffsetErr`	-23	Invalid offset value of the wavelet filter
`ippStsAnchorErr`	-22	Anchor point is outside the mask
`ippStsMaskSizeErr`	-21	Invalid mask size
`ippStsShiftErr`	-20	Shift value is negative
`ippStsSampleFactorErr`	-19	Sampling factor is less than or equal to zero
`ippStsSamplePhaseErr`	-18	Phase value is out of range, $0 \leq \text{phase} < \text{factor}$
`ippStsFIRMRFactorErr`	-17	Incorrect value of the MR FIR sampling factor

Table 2-2 Error Status Values and Messages (continued)

ippStsFIRMRPhaseErr	-16	Incorrect value of the MR FIR sampling phase
ippStsRelFreqErr	-15	Relative frequency value is out of range
ippStsFIRLenErr	-14	Length of a FIR filter is less or equal to zero
ippStsIIROrderErr	-13	Order of an IIR filter is less or equal to zero
ippStsResizeFactorErr	-12	Resize factor(s) is less or equal to zero
ippStsDivByZeroErr	-11	Attempt to divide by zero
ippStsInterpolationErr	-10	Invalid interpolation mode
ippStsMirrorFlipErr	-9	Invalid flip mode
ippStsMoment00ZeroErr	-8	Moment value M(0,0) is too small for further calculations
ippStsThreshNegLevelErr	-7	Negative value of the level in the threshold operation
ippStsContextMatchErr	-6	Context parameter does not match the operation
ippStsFftFlagErr	-5	Incorrect value of the FFT flag parameter
ippStsFftOrderErr	-4	Incorrect value of the FFT order parameter
ippStsMemAllocErr	-3	Not enough memory allocated for the operation
ippStsNullPtrErr	-2	Null pointer error
ippStsSizeErr	-1	Incorrect value of the data size
ippStsNoErr	0	No error
ippStsNoOperation	1	No operation executed
ippStsMisalignedBuf	2	Misaligned pointer encountered in the operation where it must be aligned
ippStsSqrtNegArg	3	Negative value(s) of the argument in the function Sqrt
ippStsInvByZero	4	Inf result. Zero value was encountered by the function InvThresh called with zero level
ippStsEvenMedianMaskSize	5	Even size of the Median Filter mask was replaced by an odd number
ippStsDivByZero	6	Zero value(s) of the divisor in the function Div
ippStsLnZeroArg	7	Zero value(s) of the argument in the function Ln
ippStsLnNegArg	8	Negative value(s) of the argument in the function Ln
ippStsNanArg	9	Not-a-number (NaN) met in the input

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.
All other status codes indicate warnings. When a specific case is encountered, the function execution is completed and the corresponding warning status is returned.

For example, if the integer function `ippsDiv_8u` meets an attempt to divide a positive value by zero, the function execution is not aborted. The result of the operation is set to the maximum value that can be represented by the source data type, and the function returns the warning status `ippStsDivByZero`.

Support Functions 3

This chapter describes IPP support functions that are used to:

- retrieve information about the current IPP software version
- perform specific auxiliary operations
- allocate and free memory that is needed for the operation of other IPP functions.

Version Information Function

This function returns the version number and other information about the active IPP signal processing software.

GetLibVersion

Returns information about the active version of IPP signal processing software.

```
const IppLibraryVersion* ippsGetLibVersion(void);
```

Discussion

The function `ippsGetLibVersion` returns a pointer to a static data structure `IppLibraryVersion` that contains information about the current version of the IPP for signal processing. There is no need for you to release memory referenced by the returned pointer, as it points to a static variable. The following fields of the `IppLibraryVersion` structure are available:

major	is the major number of the current library version.
minor	is the minor number of the current library version.
Name	is the name of the current library version.
Version	is the library version string.
BuildDate	is the library version actual build date.

For example, if the library version is “v1.2 Beta”, library name is “ippsm6”, and build date is “Jul 20 99”, then the fields in this structure are set as:

major = 1, *minor* = 2, *Name* = “ippsm6”,
Version = “v1.2 Beta”, *BuildDate* = “Jul 20 99”

Example 3-1 shows how to use the function `ippsGetLibVersion`.

Example 3-1 Using the ippsGetLibVersion Function

```
void libinfo(void) {
       const IppLibraryVersion* lib = ippsGetLibVersion();
       printf(“%s %s %d.%d.%d.%d\n”, lib->Name, lib->Version,
           lib->major, lib->minor, lib->majorBuild, lib->build);
}

Output:
    ippsa6 v0.0 Alpha 0.0.5.5
```

IPP Common Functions

This section describes the IPP common functions that you can use for specific purposes. For example, the function `ippGetCpuType` retrieves the type of the processor in your system and thus helps you understand if the loaded DLL is in compliance with the processor type. The function `ippGetCpuClocks` gets the current number of processor clocks, which is widely used for operations timing.The function `ippGetStatusString` gets a brief description of the status code returned by the current IPP software. The functions `ippSetFlushToZero` and `ippSetDenormAreZeros` enable or disable special processor modes.

GetCpuType

Returns a processor type.

```
IppCpuType ippGetCpuType (void);
```

Discussion

The function `ippGetCpuType` detects the processor type used in your computer system and returns an appropriate `IppCpuType` variable value. Table 3-1 lists possible values and their meaning.

Table 3-1 Processor Types Detected in IPP

Returned Variable Value	Processor Type
ippCpuPP	Pentium® processor
ippCpuPMX	Pentium® processor with MMX™ technology
ippCpuPPR	Pentium® Pro processor
ippCpuPII	Pentium® II processor
ippCpuPIII	Pentium ® III processor
ippCpuP4	Pentium® 4 processor
ippCpuITP	Itanium™ processor
ippCpuUnknown	Unknown processor

GetCpuClocks

Returns a current value of the time stamp counter (TSC) register.

```
Ipp64u ippGetCpuClocks (void);
```

Discussion

The function `ippGetCpuClocks` reads the current state of the time stamp counter (TSC) register and returns its value. A hardware exception is possible if TSC reading is not supported by the current chipset.

GetStatusString

Translates a status code into a message.

```
const char* ippGetStatusString(IppStatus StsCode);
```

Arguments

`StsCode`	Code that indicates the status type (see Table 2-2).

Discussion

The function `ippGetStatusString` returns a pointer to the text string associated with a status code of type `IppStatus`. Use this function to produce error and warning messages for users. The returned pointer is a pointer to an internal static buffer and need not be released.

A code example below shows how to use the function `ippGetStatusString`. If you call an IPP function, in this case, `ippsAdd_16s_I` with a `NULL` pointer, it returns an error code `-2`. The status information function translates this code into the corresponding message "`Null Pointer Error`".

Example 3-2 Using the ippGetStatusString Function

```
void statusinfo(void) {
       IppStatus st = ippsAddC_16s_I (3, 0, 0);
       printf("%d : %s\n", st, ippGetStatusString(st));
}

Output:
    -2, Null Pointer Error
```

SetFlushToZero

Enables or disables flush-to-zero mode.

```
IppStatus ippSetFlushToZero(int value, unsigned int* pUMask);
```

Arguments

value	Switch to set or clear the corresponding bit of the MXCSR register.
pUMask	Pointer to the current underflow exception mask; may be set to `NULL`.

Discussion

The `ippSetFlushToZero` function enables (when the *value* is not equal to 0) or disables (when the *value* is equal to 0) a flush-to-zero (FTZ) mode of processors that support SSE instructions. The FTZ mode controls the masked response to a SIMD floating-point underflow condition. The FTZ mode is provided primarily for

performance reasons. At the cost of a slight precision loss, FTZ mode enables faster execution of applications where underflows are common and rounding the underflow result to zero can be tolerated.

FTZ mode is possible only when the mask register is in a certain state. The `ippSetFlushToZero` function checks and changes this state if necessary. After disabling the FTZ mode, you can restore the initial mask register state. To do this, you must declare a variable of `unsigned integer` type in your application and point to it in the *`pUMask`* parameter of the `ippFlushToZero` function. The initial state of mask register will be saved in this location and can be restored later. If you do not need to restore the initial mask state, then the *`pUMask`* pointer may be set to `NULL`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsCpuNotSupportedErr`	Indicates an error condition when the FTZ mode is not supported by the processor.

SetDenormAreZeros

Enables or disables denormals-are-zero mode.

```
IppStatus ippSetDenormAreZeros(int value);
```

Arguments

`value`	Switch to set or clear the corresponding bit of the MXCSR register.

Discussion

The `ippSetDenormAreZeros` function enables (when the *`value`* is not equal to 0) or disables (when the *`value`* is equal to 0) the denormals-are-zero (DAZ) mode of processors that support SSE instructions. The DAZ mode controls the processor response to a SIMD floating-point denormal operand condition. When the DAZ flag is set, the processor converts all denormal source operands to zero with the sign of the original operand before performing any computations on source data. The DAZ mode

is provided to improve processor performance for applications such as streaming media processing, where rounding a denormal operand to zero does not noticeably affect the quality of the processed data.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsCpuNotSupportedErr`	Indicates an error condition when the DAZ mode is not supported by the processor.

Memory Allocation Functions

This section describes the IPP signal processing functions that allocate aligned memory blocks for data of required type or free the previously allocated memory. The size of allocated memory is specified by the number of elements allocated, *len*.

NOTE. *The only function to free the memory allocated by any of these functions is* `ippsFree()`.

Malloc

Allocates memory aligned to 32-byte boundary.

```
Ipp8u* ippsMalloc_8u(int len);
Ipp16u* ippsMalloc_16u(int len);
Ipp32u* ippsMalloc_32u(int len);
Ipp8s* ippsMalloc_8s(int len);
Ipp16s* ippsMalloc_16s(int len);
Ipp32s* ippsMalloc_32s(int len);
Ipp64s* ippsMalloc_64s(int len);
```

```
Ipp32f* ippsMalloc_32f(int len);
Ipp64f* ippsMalloc_64f(int len);
Ipp8sc* ippsMalloc_8sc(int len);
Ipp16sc* ippsMalloc_16sc(int len);
Ipp32sc* ippsMalloc_32sc(int len);
Ipp64sc* ippsMalloc_64sc(int len);
Ipp32fc* ippsMalloc_32fc(int len);
Ipp64fc* ippsMalloc_64fc(int len);
```

Arguments

len	Number of elements to allocate.

Discussion

The function ippsMalloc allocates memory block aligned to a 32-byte boundary for elements of different data types.

Return Value

The return value of ippsMalloc is a pointer to an aligned memory block. If no memory is available in the system, then the NULL value is returned. To free this block, use the function ippsFree.

Free

Frees memory allocated by the function ippsMalloc.

```
void ippsFree(void* ptr);
```

Arguments

ptr	Pointer to a memory block to be freed. The memory block pointed to with *ptr* has been allocated by the function ippsMalloc.

Discussion

The function `ippsFree` frees the aligned memory block allocated by the function `ippsMalloc`.

NOTE. *You can not use the function* `ippsFree` *to free memory allocated by standard functions like* `malloc` *or* `calloc`*, nor can the memory allocated by* `ippsMalloc` *be freed by* `free`*.*

Vector Initialization Functions

4

This chapter describes IPP functions that initialize vectors with either constants, the contents of other vectors, or the generated signals.

Vector Initialization Functions

This section describes functions that initialize the values of vector elements. All vector elements can be initialized to a common zero or another specified value. They can also be initialized to respective values of a second vector elements.

Copy

Copies the contents of one vector into another.

```
IppStatus ippsCopy_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsCopy_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsCopy_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsCopy_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCopy_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsCopy_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len );
IppStatus ippsCopy_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector used to initialize *pDst*.
pDst	Pointer to the destination vector to be initialized.
len	Number of elements to copy.

Discussion

The function ippsCopy copies the first *len* elements from a source vector *pSrc* into a destination vector *pDst*.

Example 4-1 shows how to use the function ippsCopy.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 4-1 Using the ippsCopy Function

```
IppStatus copy(void) {
      char src[] = “to be copied\0”;
      char dst[256];
      return ippsCopy_8u(src, dst, strlen(src)+1);
}
```

Set

Initializes vector elements to a specified common value.

```
IppStatus ippsSet_8u(Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsSet_16s(Ipp16s val, Ipp16s* pDst, int len);
IppStatus ippsSet_16sc(Ipp16sc val, Ipp16sc* pDst, int len);
IppStatus ippsSet_32s(Ipp32s val, Ipp32s* pDst, int len);
IppStatus ippsSet_32f(Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsSet_32sc(Ipp32sc val, Ipp32sc* pDst, int len);
IppStatus ippsSet_32fc(Ipp32fc val, Ipp32fc* pDst, int len);
```

```
IppStatus ippsSet_64s(Ipp64s val, Ipp64s* pDst, int len);
IppStatus ippsSet_64f(Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsSet_64sc(Ipp64sc val, Ipp64sc* pDst, int len);
IppStatus ippsSet_64fc(Ipp64fc val, Ipp64fc* pDst, int len);
```

Arguments

pDst	Pointer to the vector to be initialized.
len	Number of elements to initialize.
val	Value used to initialize the vector *pDst*.

Discussion

The function `ippsSet` initializes the first *len* elements of the real or complex vector *pDst* to contain the same value *val* .

Example 4-2 shows how to use the function `ippsSet`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example 4-2 Using the ippsSet Function

```
IppStatus set(void) {
      char src[] = “set”;
      return ippsSet_8u(‘0’, src, strlen(src));
}
```

Zero

Initializes a vector to zero.

```
IppStatus ippsZero_8u(Ipp8u* pDst, int len);
IppStatus ippsZero_16s(Ipp16s* pDst, int len);
IppStatus ippsZero_16sc(Ipp16sc* pDst, int len);
IppStatus ippsZero_32f(Ipp32f* pDst, int len);
IppStatus ippsZero_32fc(Ipp32fc* pDst, int len);
IppStatus ippsZero_64f(Ipp64f* pDst, int len);
IppStatus ippsZero_64fc(Ipp64fc* pDst, int len);
```

Arguments

pDst	Pointer to the vector to be initialized to zero.
len	Number of elements to initialize.

Discussion

The function `ippsZero` initializes the first *len* elements of the vector *pDst* to 0. If *pDst* is a complex vector, both real and imaginary parts are zeroed.

Example 4-3 shows how to use the function `ippsZero`.

Return Value

`ippStsNoErr`	Indicates no error
`ippStsNullPtrErr`	Indicates an error when the *pDst* pointer is `NULL`
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0

Example 4-3 Using the ippsZero Function

```
IppStatus zero(void) {
   char src[] = “zero”;
   return ippsZero_8u(src, strlen(src));
}
```

Sample-Generating Functions

This section describes IPP functions which generate tone samples, triangle samples, pseudo-random samples with uniform distribution, and pseudo-random samples with Gaussian distribution, as well as special test samples.

Tone-Generating Functions

The functions described below generate a tone (or “sinusoid”) of a given frequency, phase, and magnitude. Tones are fundamental building blocks for analog signals. Thus, sampled tones are extremely useful in signal processing systems as test signals and as building blocks for more complex signals.

The use of tone functions is preferable against the analogous C math library’s `sin()` function for many applications, because IPP functions can use information retained from the computation of the previous sample to compute the next sample much faster than standard `sin()` or `cos()`.

Tone

Generates a tone with a given frequency, phase, and magnitude.

```
IppStatus ippsTone_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn,
     float rfreq, float* pPhase, IppHintAlgorithm hint);
```

```
IppStatus ippsTone_Direct_16sc(Ipp16sc* pDst, int len, Ipp16s magn,
     float rfreq, float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_32f(Ipp32f* pDst, int len, float magn,
     float rfreq, float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_32fc(Ipp32fc* pDst, int len, float magn,
     float rfreq, float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_64f(Ipp64f* pDst, int len, double magn,
     double rfreq, double* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_64fc(Ipp64fc* pDst, int len, double magn,
     double rfreq, double* pPhase, IppHintAlgorithm hint);
```

Arguments

magn	Magnitude of the tone; that is, the maximum value attained by the wave.
pPhase	Pointer to the phase of the tone relative to a cosine wave. It must be between 0.0 and 2π. The returned value may be used to compute the next continuous data block.
rfreq	Frequency of the tone relative to the sampling frequency. It must be between 0.0 and 0.5.
pDst	Pointer to the array which stores the samples.
len	Number of samples to be computed.
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."

Discussion

The function ippsTone generates the tone with the specified frequency *rfreq*, phase *pPhase*, and magnitude *magn*. The function computes *len* samples of the tone, and stores them in the array *pDst*. For real tones, each generated value *x(n)* is defined as:

$x(n) = magn \cdot \cos(2\pi n \cdot rfreq + phase)$

For complex tones, *x(n)* is defined as:

$x(n) = magn \cdot (\cos(2\pi n \cdot rfreq + phase) + j \cdot \sin(2\pi n \cdot rfreq + phase))$

The *hint* argument suggests using specific code, which provides for either fast but less accurate calculation, or more accurate but slower execution. The values you can enter for the *hint* argument are listed in Table 7-2.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pPhase* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsToneMagnErr	Indicates an error when *magn* is negative.
ippStsToneFreqErr	Indicates an error when *rfreq* is negative, or greater than or equal to 0.5.
ippStsTonePhaseErr	Indicate an error when the *pPhase* value is negative, or greater than or equal to IPP_2PI.

Triangle-Generating Functions

This section describes the function that generates a periodic signal with a triangular wave form (referred to as “triangle”) of a given frequency, phase, magnitude, and asymmetry.

Application Notes

A real periodic signal with triangular wave form $x[n]$ (referred to as a real triangle) of a given frequency *rfreq*, phase value *phase*, magnitude *magn*, and asymmetry h is defined as follows:

$$x[n] = magn \cdot \mathbf{ct}_h(2\pi \cdot rfreq \cdot n + phase),\ n = 0, 1, 2, \ldots$$

A complex periodic signal with triangular wave form *x[n]* (referred to as a complex triangle) of a given frequency *rfreq*, phase value *phase*, magnitude *mag*, and asymmetry h is defined as follows:

$$x[n] = magn \cdot (\mathbf{ct}_h(2\pi \cdot rfreq \cdot n + phase) + j \cdot \mathbf{st}_h(2\pi \cdot rfreq \cdot n + phase)),\ n = 0, 1, 2, \ldots$$

The $\mathbf{ct}_h()$ function is determined as follows:

$H = \pi + h$

$$\mathbf{ct}_h(\alpha) = \begin{cases} -\frac{2}{H} \cdot \left(\alpha - \frac{H}{2}\right), & 0 \le \alpha \le H \\ \frac{2}{2\pi - H} \cdot \left(\alpha - \frac{2\pi + H}{2}\right), & H \le \alpha \le 2\pi \end{cases}$$

$$\mathbf{ct}_h(\alpha + k \cdot 2\pi) = \mathbf{ct}_h(\alpha), \; k = 0, \pm 1, \pm 2, \ldots$$

When $H = \pi$, asymmetry $h = 0$, and function $\mathbf{ct}_h()$ is symmetric and a triangular analog of the cos() function. Note the following equations:

$$\mathbf{ct}_h(H/2 + k \cdot \pi) = 0, \; k = 0, \pm 1, \pm 2, \ldots$$

$$\mathbf{ct}_h(k \cdot 2\pi) = 1, \; k = 0, \pm 1, \pm 2, \ldots$$

$$\mathbf{ct}_h(H + k \cdot 2\pi) = -1, \; k = 0, \pm 1, \pm 2, \ldots$$

The $\mathbf{st}_h()$ function is determined as follows:

$$\mathbf{st}_h(\alpha) = \begin{cases} \frac{2}{2\pi - H} \cdot \alpha, & 0 \le \alpha \le \frac{2\pi - H}{2} \\ -\frac{2}{H} \cdot (\alpha - \pi), & \frac{2\pi - H}{2} \le \alpha \le \frac{2\pi + H}{2} \\ \frac{2}{2\pi - H} \cdot (\alpha - 2\pi), & \frac{2\pi + H}{2} \le \alpha \le 2\pi \end{cases}$$

$$\mathbf{st}_h(\alpha + k \cdot 2\pi) = \mathbf{st}_h(\alpha), \; k = 0, \pm 1, \pm 2, \ldots$$

When $H = \pi$, asymmetry $h = 0$, and function $\mathbf{st}_h()$ is a triangular analog of a sine function. Note the following equations:

$$\mathbf{st}_h(\alpha) = \mathbf{ct}_h(\alpha + (3\pi + h)/2)$$

$$\mathbf{st}_h(\pi k) = 0, \; k = 0, \pm 1, \pm 2, \ldots$$

$$\mathbf{st}_h((\pi - h)/2 + 2\pi k) = 1, \; k = 0, \pm 1, \pm 2, \ldots$$

$$\mathbf{st}_h((3\pi + h)/2 + 2\pi k) = -1, \; k = 0, \pm 1, \pm 2, \ldots$$

Triangle

Generates a triangle with a given frequency, phase, and magnitude.

```
IppStatus ippsTriangle_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn,
     float rfreq, float asym, float* pPhase);
IppStatus ippsTriangle_Direct_16sc(Ipp16sc* pDst, int len, Ipp16s magn,
     float rfreq, float asym, float* pPhase);
IppStatus ippsTriangle_Direct_32f(Ipp32f* pDst, int len, float magn,
     float rfreq, float asym, float* pPhase);
IppStatus ippsTriangle_Direct_32fc(Ipp32fc* pDst, int len, float magn,
     float rfreq, float asym, float* pPhase);
IppStatus ippsTriangle_Direct_64f(Ipp64f* pDst, int len, double magn,
     double rfreq, double asym, double* pPhase);
IppStatus ippsTriangle_Direct_64fc(Ipp64fc* pDst, int len, double magn,
     double rfreq, double asym, double* pPhase);
```

Arguments

rfreq	Frequency of the triangle relative to the sampling frequency. It must be between 0.0 and 0.5.
pPhase	Pointer to the phase of the triangle relative to a cosine triangular analog wave. It must be between 0.0 and 2π. The returned value may be used to compute the next continuous data block.
magn	Magnitude of the triangle; that is, the maximum value attained by the wave.
asym	Asymmetry h of a triangle. It must be between $-\pi$ and π. If h=0, then the triangle is symmetric and a direct analog of a tone.
pDst	Pointer to the array which stores the samples.
len	Number of samples to be computed.

Discussion

The function ippsTriangle generates the triangle with the specified frequency *rfreq*, phase pointed by *pPhase*, and magnitude *magn*. The function computes *len* samples of the triangle, and stores them in the array *pDst*. For real triangle, *x(n)* is defined as:

$$x[n] = magn \cdot \mathbf{ct}_h(2\pi \cdot rfreq \cdot n + phase), \quad n = 0, 1, 2, \ldots$$

For complex triangles, *x(n)* is defined as:

$$x[n] = magn \cdot (\mathbf{ct}_h(2\pi \cdot rfreq \cdot n + phase) + j \cdot \mathbf{st}_h(2\pi \cdot rfreq \cdot n + phase)), \quad n = 0, 1, 2, \ldots$$

See "Application Notes" on page 4-7 for the definition of functions $\mathbf{ct}_h$ and $\mathbf{st}_h$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pPhase* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsTrnglMagnErr	Indicates an error when *magn* is negative.
ippStsTrnglFreqErr	Indicates an error when *rfreq* is negative, or greater than or equal to 0.5.
ippStsTrnglPhaseErr	Indicate an error when the *pPhase* value is negative, or greater than or equal to IPP_2PI.
ippStsTrnglAsymErr	Indicate an error when *asym* is less than -IPP_PI, or greater than or equal to IPP_PI.

Uniform Distribution Functions

This section describes the functions that generate pseudo-random samples with uniform distribution.

RandUniformInitAlloc

Initializes a noise generator with uniform distribution.

```
IppStatus ippsRandUniformInitAlloc_8u(IppsRandUniState_8u**
     pRandUniState, Ipp8u low, Ipp8u high, unsigned int seed);
IppStatus ippsRandUniformInitAlloc_16s(IppsRandUniState_16s**
     pRandUniState, Ipp16s low, Ipp16s high, unsigned int seed);
IppStatus ippsRandUniformInitAlloc_32f(IppsRandUniState_32f**
     pRandUniState, Ipp32f low, Ipp32f high, unsigned int seed);
```

Arguments

pRandUniState	Pointer to the structure containing parameters for the generator of noise.
low	Lower bound of the uniform distribution range.
high	Upper bound of the uniform distribution range.
seed	Seed value used by the pseudo-random number generation algorithm.

Discussion

The `ippsRandUniformInitAlloc` function allocates memory and initializes the pseudo-random generator state *pRandUniState*. The uniform distribution range is specified by the lower and upper bounds *low* and *high*, respectively.

Return Value

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pRandUniState* pointer is NULL.
ippStsMemAllocErr	Indicates an error when there is not enough memory for the operation.

RandUniformFree

Closes the uniform distribution generator state.

```
IppStatus ippsRandUniformFree_8u(IppsRandUniState_8u* pRandUniState);
IppStatus ippsRandUniformFree_16s(IppsRandUniState_16s* pRandUniState);
IppStatus ippsRandUniformFree_32f(IppsRandUniState_32f* pRandUniState);
```

Arguments

pRandUniState	Pointer to the structure containing parameters for the generator of noise.

Discussion

The ippsRandUniformFree function closes the noise generator state *pRandUniState* by freeing all memory allocated by the ippsRandUniInitAlloc function.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pRandUniState* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

RandUnifrom

Generates the pseudo-random samples with a uniform distribution.

```
IppStatus ippsRandUniform_8u(Ipp8u* pDst,  int len,
     IppsRandUniState_8u* pRandUniState);
IppStatus ippsRandUniform_16s(Ipp16s* pDst, int len,
     IppsRandUniState_16s* pRandUniState);
IppStatus ippsRandUniform_32f(Ipp32f* pDst, int len,
     IppsRandUniState_32f* pRandUniState);
```

Arguments

pDst	Pointer to the array which stores the samples.
len	Number of samples to be computed.
pRandUniState	Pointer to the structure containing parameters for the generator of noise.

Discussion

The ippsRandUniform function generates *len* pseudo-random samples with a uniform distribution and stores them in the array *pDst*. Initial parameters of the generator are set in the generator state structure *pRandUniState*.
Before calling ippsRandUniform, you must initialize the generator state by calling ippsRandUniformInitAlloc function.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pRandUniState* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

RandUniform_Direct

Generates the pseudo-random samples with a uniform distribution in direct mode.

```
IppStatus ippsRandUniform_Direct_16s(Ipp16s* pDst, int len, Ipp16s low,
      Ipp16s high, unsigned int* pSeed);
IppStatus ippsRandUniform_Direct_32f(Ipp32f* pDst, int len, Ipp32f low,
      Ipp32f high, unsigned int* pSeed);
IppStatus ippsRandUniform_Direct_64f(Ipp64f* pDst, int len, Ipp64f low,
      Ipp64f high, unsigned int* pSeed);
```

Arguments

pSeed	Pointer to the seed value used by the pseudo-random number generation algorithm.
low	Lower bound of the uniform distribution range.
high	Upper bound of the uniform distribution range.
pDst	Pointer to the array which stores the samples.
len	Number of samples to be computed.

Discussion

The function `ippsRandUniform_Direct` generates *len* pseudo-random samples with a uniform distribution and stores them in the array *pDst*. This function does not require to initialize the generator state structure in advance. All parameters of the pseudo-random number generator are set directly in the function.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pSeed* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Gaussian Distribution Functions

This section describes the function that generates pseudo-random samples with Gaussian distribution.

RandGaussInitAlloc

Initializes a noise generator with Gaussian distribution.

```
IppStatus ippsRandGaussInitAlloc_8u(IppsRandGaussState_8u**
     pRandGaussState, Ipp8u mean, Ipp8u stdDev, unsigned int seed);
IppStatus ippsRandGaussInitAlloc_16s(IppsRandGaussState_16s**
     pRandGaussState, Ipp16s mean, Ipp16s stdDev, unsigned int seed);
IppStatus ippsRandGaussInitAlloc_32f(IppsRandGaussState_32f**
     pRandGaussState, Ipp32f mean, Ipp32f stdDev, unsigned int seed);
```

Arguments

pRandGaussState	Pointer to the structure containing parameters for the generator of noise.
mean	Mean of the Gaussian distribution.
stdDev	Standard deviation of the Gaussian distribution.
seed	Seed value used by the pseudo-random number generator algorithm.

Discussion

The `ippsRandGaussInitAlloc` function allocates memory and initializes the pseudo-random generator state structure *pRandGaussState*. This structure contains parameters of the required noise generator that are specified by the *mean*, *stdDev* and *seed* values.

Return Value

ppStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pRandGaussState* pointer is NULL.
ippStsMemAllocErr	Indicates an error when there is not enough memory for the operation.

RandGaussFree

Closes the Gaussian distribution generator state.

```
IppStatus ippsRandGaussFree_8u(IppsRandGaussState_8u* pRandGaussState);
IppStatusippsRandGaussFree_16s(IppsRandGaussState_16s*pRandGaussState);
IppStatusippsRandGaussFree_32f(IppsRandGaussState_32f*pRandGaussState);
```

Arguments

pRandGaussState	Pointer to the structure containing parameters for the generator of noise.

Discussion

The ippsRandGaussFree function closes the noise generator state *pRandGaussState* by freeing all memory allocated by the ippsRandGaussInitAlloc function.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pRandGaussState* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

RandGauss

Generates the pseudo-random samples with a Gaussian distribution.

```
IppStatus ippsRandGauss_8u(Ipp8u* pDst,  int len,
     IppsRandGaussState_8u*  pRandGaussState);
IppStatus ippsRandGauss_16s(Ipp16s* pDst, int len,
     IppsRandGaussState_16s* pRandGaussState);
IppStatus ippsRandGauss_32f(Ipp32f* pDst, int len,
     IppsRandGaussState_32f* pRandGaussState);
```

Arguments

pDst	Pointer to the array which stores the samples.
len	Number of samples to be computed.
pRandGaussState	Pointer to the structure containing parameters of the noise generator.

Discussion

The ippsRandGauss function generates *len* pseudo-random samples with a Gaussian distribution and stores them in the array *pDst*. The initial parameters of the generator are set in the generator state structure *pRandGaussState*.
Before calling ippsRandGauss, you must initialize the generator state by calling ippsRandGaussInitAlloc function.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pRandGaussState* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

RandGauss_Direct

Generates pseudo-random samples with a Gaussian distribution in the direct mode.

```
IppStatus ippsRandGauss_Direct_16s(Ipp16s* pDst, int len, Ipp16s mean,
      Ipp16s stdev, unsigned int* pSeed);
IppStatus ippsRandGauss_Direct_32f(Ipp32f* pDst, int len, Ipp32f mean,
      Ipp32f stdev, unsigned int* pSeed);
IppStatus ippsRandGauss_Direct_64f(Ipp64f* pDst, int len, Ipp64f mean,
      Ipp64f stdev, unsigned int* pSeed);
```

Arguments

pDst	Pointer to the array which stores the samples.
pSeed	Pointer to the seed value used by the pseudo-random number generation algorithm.
len	Number of samples to be computed.
mean	Mean of the Gaussian distribution.
stdev	Standard deviation of the Gaussian distribution.

Discussion

The function ippsRandGauss generates *len* pseudo-random samples with a Gaussian distribution, and stores them in the array *pDst*.This function does not require to initialize the generator state structure in advance. All parameters of the pseudo-random number generator are set directly in the function.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSeed* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Special Vector Functions

The functions described in this section create special vectors that can be used as a test signals to examine the effect of applying different signal processing functions.

VectorJaehne

Creates a Jaehne vector.

```
IppStatus ippsVectorJaehne_8u(Ipp8u* pDst, int len, Ipp8u magn);
IppStatus ippsVectorJaehne_8s(Ipp8s* pDst, int len, Ipp8s magn);
IppStatus ippsVectorJaehne_16u(Ipp16u* pDst, int len, Ipp16u magn);
IppStatus ippsVectorJaehne_16s(Ipp16s* pDst, int len, Ipp16s magn);
IppStatus ippsVectorJaehne_32u(Ipp32u* pDst, int len, Ipp32u magn);
IppStatus ippsVectorJaehne_32s(Ipp32s* pDst, int len, Ipp32s magn);
IppStatus ippsVectorJaehne_32f(Ipp32f* pDst, int len, Ipp32f magn);
IppStatus ippsVectorJaehne_64f(Ipp64f* pDst, int len, Ipp64f magn);
```

Arguments

pDst	Pointer to the destination vector.
len	Number of elements in the vector.
magn	Magnitude of the signal to be generated.

Discussion

The function `ippsVectorJaehne` creates a Jaehne vector and stores the result in *pDst*. The magnitude *magn* must be positive. The function generates the sinusoid with a variable frequency. The computation is performed as follows:

$$pDst[n] = magn \cdot \sin\left(\frac{0.5\pi n^2}{len}\right), \quad 0 \le n < len$$

Example 4-4 shows how to use the function `ippsVectorJaehne`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcDst`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`ippStsJaehneErr`	Indicates an error when *`magn`* is negative.

Example 4-4 Using the ippsVectorJaehne Function

```
IppStatus Jaehne (void) {
    Ipp16s buf[100] ;
    return ippsVectorJaehne_16s ( buf, 100, 255 );
}
```

VectorRamp

Creates a ramp vector.

```
IppStatus ippsVectorRamp_8u(Ipp8u* pDst, int len, float offset,
      float slope);
IppStatus ippsVectorRamp_8s(Ipp8s* pDst, int len, float offset,
      float slope);
IppStatus ippsVectorRamp_16u(Ipp16u* pDst, int len, float offset,
      float slope);
IppStatus ippsVectorRamp_16s(Ipp16s* pDst, int len, float offset,
      float slope);
IppStatus ippsVectorRamp_32u(Ipp32u* pDst, int len, float offset,
      float slope);
IppStatus ippsVectorRamp_32s(Ipp32s* pDst, int len, float offset,
      float slope);
IppStatus ippsVectorRamp_32f(Ipp32f* pDst, int len, float offset,
      float slope);
```

```
IppStatus ippsVectorRamp_64f(Ipp64f* pDst, int len, float offset,
     float slope);
```

Arguments

pDst	Pointer to the destination vector.
len	Number of elements in the vector.
offset	Offset value.
slope	Slope coefficient.

Discussion

The function ippsVectorRamp creates a ramp vector and stores the result in *pDst*. The destination vector elements are computed according to the following formula:

pDst[i] = *offset* + *slope***n*, $0 \le n < len$.

Note that linear transform coefficients *offset* and *slope* have floating-point values for all function flavors.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Essential Vector Functions 5

This chapter describes IPP functions that perform logical and shift, arithmetic, conversion, companding, windowing, and statistical operations.

Logical and Shift Functions

This section describes the IPP signal processing functions that perform logical and shift operations on vectors. Logical and shift functions are only defined for integer arguments.

For binary logical operations AND, OR and XOR, the following functions are provided:
`AndC`, `OrC`, `XorC` for vector-scalar operations
`And`, `Or`, `Xor` for vector-vector operations.

AndC

Computes the bitwise AND of a scalar value and each element of a vector.

```
IppStatus ippsAndC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
     int len);
IppStatus ippsAndC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst,
     int len);
IppStatus ippsAndC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);
IppStatus ippsAndC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
```

Arguments

val	Input scalar value.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsAndC computes the bitwise AND of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of ippsAndC compute the bitwise AND of a scalar value *val* and each element of the vector *pSrcDst* and store the result in *pSrcDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

And

Computes the bitwise AND of two vectors.

```
IppStatus ippsAnd_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
     Ipp8u* pDst, int len);
IppStatus ippsAnd_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
     Ipp16u* pDst, int len);
IppStatus ippsAnd_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsAnd_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
```

Arguments

pSrc1, pSrc2	Pointers to the two source vectors.
pDst	Pointer to the destination vector.
pSrc	Pointer to the source vector for the in-place operation.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsAnd computes the bitwise AND of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

The in-place flavors of ippsAnd compute the bitwise AND of the corresponding elements of the vectors *pSrc* and *pSrcDst* and store the result in the vector *pSrcDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

OrC

Computes the bitwise OR of a scalar value and each element of a vector.

```
IppStatus ippsOrC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsOrC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);
IppStatus ippsOrC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);
IppStatus ippsOrC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
```

Arguments

val	Input scalar value.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsOrC computes the bitwise OR of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of ippsOrC compute the bitwise OR of a scalar value *val* and each element of the vector *pSrcDst* and store the result in *pSrcDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Or

Computes the bitwise OR of two vectors.

```
IppStatus ippsOr_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
     Ipp8u* pDst, int len);
IppStatus ippsOr_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
     Ipp16u* pDst, int len);
IppStatus ippsOr_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsOr_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
```

Arguments

pSrc1, pSrc2	Pointers to the two source vectors.
pDst	Pointer to the destination vector.
pSrc	Pointer to the source vector for the in-place operation.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsOr computes the bitwise OR of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

The in-place flavors of ippsOr compute the bitwise OR of the corresponding elements of the vectors *pSrc* and *pSrcDst* and store the result in the vector *pSrcDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

XorC

Computes the bitwise XOR of a scalar value and each element of a vector.

```
IppStatus ippsXorC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsXorC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);
IppStatus ippsXorC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);
IppStatus ippsXorC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
```

Arguments

val	Input scalar value.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsXorC computes the bitwise XOR of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of ippsXorC compute the bitwise XOR of a scalar value *val* and each element of the vector *pSrcDst* and store the result in *pSrcDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Xor

Computes the bitwise XOR of two vectors.

```
IppStatus ippsXor_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
     Ipp8u* pDst, int len);
IppStatus ippsXor_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
     Ipp16u* pDst, int len);
IppStatus ippsXor_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsXor_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
```

Arguments

pSrc1, pSrc2	Pointers to the two source vectors.
pDst	Pointer to the destination vector.
pSrc	Pointer to the source vector for the in-place operation.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsXor computes the bitwise XOR of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

The in-place flavors of ippsXor compute the bitwise XOR of the corresponding elements of the vectors *pSrc* and *pSrcDst* and store the result in the vector *pSrcDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Not

Computes the bitwise NOT of the vector elements.

```
IppStatus ippsNot_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsNot_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsNot_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsNot_16u_I(Ipp16u* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsNot computes the bitwise NOT of the corresponding elements of the vectors *pSrc*, and stores the result in the vector *pDst*.

The in-place flavors of ippsNot compute the bitwise NOT of the corresponding elements of the vector *pSrcDst* and store the result in the vector *pSrcDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

LShiftC

Shifts bits in vector elements to the left.

```
IppStatus ippsLShiftC_8u(const Ipp8u* pSrc, int val, Ipp8u* pDst, int len);
IppStatus ippsLShiftC_16s(const Ipp16s* pSrc, int val, Ipp16s* pDst, int len);
IppStatus ippsLShiftC_16u(const Ipp16u* pSrc, int val, Ipp16u* pDst, int len);
IppStatus ippsLShiftC_32s(const Ipp32s* pSrc, int val, Ipp32s* pDst, int len);
IppStatus ippsLShiftC_8u_I(int val, Ipp8u* pSrcDst, int len);
IppStatus ippsLShiftC_16u_I(int val, Ipp16u* pSrcDst, int len);
IppStatus ippsLShiftC_16s_I(int val, Ipp16s* pSrcDst, int len);
IppStatus ippsLShiftC_32s_I(int val, Ipp32s* pSrcDst, int len);
```

Arguments

val	Number of bits by which the function shifts each element of the vector *pSrc* or *pSrcDst*.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsLShiftC shifts each element of the vector *pSrc* by *val* bits to the left, and stores the result in *pDst*.

The in-place flavors of ippsLShiftC shift each element of the vector *pSrcDst* by *val* bits to the left and store the result in *pSrcDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

RShiftC

Shifts bits in vector elements to the right.

```
IppStatus ippsRShiftC_8u(const Ipp8u* pSrc, int val, Ipp8u* pDst, int len);
IppStatus ippsRShiftC_16s(const Ipp16s* pSrc, int val, Ipp16s* pDst, int len);
IppStatus ippsRShiftC_16u(const Ipp16u* pSrc, int val, Ipp16u* pDst, int len);
IppStatus ippsRShiftC_32s(const Ipp32s* pSrc, int val, Ipp32s* pDst, int len);
```

```
IppStatus ippsRShiftC_8u_I(int val, Ipp8u* pSrcDst, int len);
IppStatus ippsRShiftC_16u_I(int val, Ipp16u* pSrcDst, int len);
IppStatus ippsRShiftC_16s_I(int val, Ipp16s* pSrcDst, int len);
IppStatus ippsRShiftC_32s_I(int val, Ipp32s* pSrcDst, int len);
```

Arguments

val	Number of bits by which the function shifts each element of the vector *pSrc* or *pSrcDst*.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsRShiftC shifts each element of the vector *pSrc* by *val* bits to the right, and stores the result in *pDst*.

The in-place flavors of ippsRShiftC shift each element of the vector *pSrcDst* by *val* bits to the right and store the result in *pSrcDst*.

Note that the arithmetic shift is realized for signed data, and the logical shift for unsigned data.

Example 5-1 shows how the logical and shift functions can be used in the saturate operation. The data are converted to the unsigned char range [0...255].

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-1 Using the Logical and Shift Functions

```
void saturate(void) {
      Ipp16s x[8] = {1000, -257, 127, 4, 5, 0, 7, 8}, lo[8], hi[8];
      IppStatus status = ippsNot_16u((Ipp16u*)x, (Ipp16u*)lo, 8);
      ippsRShiftC_16s_I(15, lo, 8);
      ippsCopy_16s(x, hi, 8);
      ippsSubCRev_16s_ISfs(255, hi, 8, 0);
      ippsRShiftC_16s_I(15, hi, 8);
      ippsAnd_16u_I((Ipp16u*)lo, (Ipp16u*)x, 8);
      ippsOr_16u_I((Ipp16u*)hi, (Ipp16u*)x, 8);
      ippsAndC_16u_I(255, (Ipp16u*)x, 8);
      printf_16s("saturate =", x, 8, status);
}
Output:
    saturate =  255 0 127 4 5 0 7 8
```

Arithmetic Functions

This section describes the IPP signal processing functions that perform vector arithmetic operations on vectors. The arithmetic functions include basic element-wise arithmetic operations between vectors, as well as more complex calculations such as computing absolute values, square and square root, natural logarithm and exponential of vector elements.

IPP software provides two versions of each function. One version performs the operation in-place, while the other stores the results of the operation in a different destination vector, that is, executes an out-of-place operation.

AddC

Adds a constant value to each element of a vector.

```
IppStatus ippsAddC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst,
      int len);
IppStatus ippsAddC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst,
      int len);
IppStatus ippsAddC_32fc(const Ipp32fc* pSrc, Ipp32fc val,Ipp32fc* pDst,
      int len);
IppStatus ippsAddC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst,
      int len);
IppStatus ippsAddC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsAddC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsAddC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsAddC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsAddC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsAddC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
      int len, int scaleFactor);
IppStatus ippsAddC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s*
      pDst, int len, int scaleFactor);
IppStatus ippsAddC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s*
      pDst, int len, int scaleFactor);
IppStatus ippsAddC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc*
      pDst, int len, int scaleFactor);
IppStatus ippsAddC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc*
      pDst, int len, int scaleFactor);
IppStatus ippsAddC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len,
      int scaleFactor);
IppStatus ippsAddC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len,
      int scaleFactor);
IppStatus ippsAddC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len,
      int scaleFactor);
```

```
IppStatus ippsAddC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsAddC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len,
     int scaleFactor);
```

Arguments

val	Scalar value used to increment each element of the vector *pSrc* or *pSrcDst*.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector which stores the result of the addition *pSrc[n]* + *val*.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsAddC adds a value *val* to each element of the source vector *pSrc*, and stores the result in the destination vector *pDst*.

The in-place flavors of ippsAddC add a value *val* to each element of the vector *pSrcDst*, and store the result in *pSrcDst*.

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Add

Adds the elements of two vectors.

```
IppStatus ippsAdd_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp16s* pDst, int len);
IppStatus ippsAdd_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
      Ipp16u* pDst, int len);
IppStatus ippsAdd_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2,
      Ipp32u* pDst, int len);
IppStatus ippsAdd_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
      Ipp32f* pDst, int len);
IppStatus ippsAdd_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
      Ipp64f* pDst, int len);
IppStatus ippsAdd_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
      Ipp32fc* pDst, int len);
IppStatus ippsAdd_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
      Ipp64fc* pDst, int len);
IppStatus ippsAdd_8u16u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
      Ipp16u* pDst, int len);
IppStatus ippsAdd_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp32f* pDst, int len);
IppStatus ippsAdd_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsAdd_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsAdd_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsAdd_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsAdd_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsAdd_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
      Ipp8u* pDst, int len, int scaleFactor);
IppStatus ippsAdd_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsAdd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
      Ipp32s* pDst, int len, int scaleFactor);
```

```
IppStatus ippsAdd_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
     Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsAdd_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2,
     Ipp32sc* pDst, int len, int scaleFactor);
IppStatus ippsAdd_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsAdd_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsAdd_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len,
      int scaleFactor);
IppStatus ippsAdd_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int
     len, int scaleFactor);
IppStatus ippsAdd_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int
     len, int scaleFactor)
```

Arguments

pSrc1, pSrc2	Pointers to the vectors whose elements are to be added together.
pDst	Pointer to the destination vector which stores the result of the addition *pSrc2[n]* + *pSrc1[n]*.
pSrc	Pointer to the source vector whose elements are to be added to the elements of *pSrcDst* in-place.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsAdd adds the elements of the vector *pSrc1* to the elements of the vector *pSrc2*, and stores the result in *pDst*.

The in-place flavors of ippsAdd add the elements of the vector *pSrc* to the elements of the vector *pSrcDst* and store the result in *pSrcDst*.

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Example 5-2 shows that overflow does not occur while adding big numbers due to the scaling operation.

Return Value

ippStsNoErr	Indicates no error
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0

Example 5-2 Using the ippsAdd Function

```
IppStatus add(void) {
      Ipp16s x[4] = {-1, 32767, 2, 30000};
      IppStatus st = ippsAdd_16s_ISfs(x, x, 4, 1);
      printf_16s("add =", x, 4, st);
      return st;
}
Output:
    add = -1 32767 2 30000
```

MulC

Multiplies each elements of a vector by a constant value.

```
IppStatus ippsMulC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst,
     int len);
IppStatus ippsMulC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst,
     int len);
```

```
IppStatus ippsMulC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc*
     pDst, int len);
IppStatus ippsMulC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc*
     pDst, int len);
IppStatus ippsMulC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsMulC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsMulC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsMulC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsMulC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsMulC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
     int len, int scaleFactor);
IppStatus ippsMulC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s*
     pDst, int len, int scaleFactor);
IppStatus ippsMulC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s*
     pDst, int len, int scaleFactor);
IppStatus ippsMulC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc*
     pDst, int len, int scaleFactor);
IppStatus ippsMulC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc*
     pDst, int len, int scaleFactor);
IppStatus ippsMulC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsMulC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsMulC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsMulC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsMulC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len,
     int scaleFactor);
```

Arguments

val	The scalar value used to multiply each element of the vector *pSrc* or *pSrcDst*.
pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector which stores the result of the multiplication *pSrc[n] * val*.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	The number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsMulC multiplies each element of the vector *pSrc* by a value *val* and stores the result in *pDst*.

The in-place flavors of ippsMulC multiply each element of the vector *pSrcDst* by a value *val* and store the result in *pSrcDst*.

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Mul

Multiplies the elements of two vectors.

```
IppStatus ippsMul_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp16s* pDst, int len);
IppStatus ippsMul_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
      Ipp32f* pDst, int len);
```

```
IppStatus ippsMul_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
     Ipp64f* pDst, int len);

IppStatus ippsMul_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
     Ipp32fc* pDst, int len);

IppStatus ippsMul_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
     Ipp64fc* pDst, int len);

IppStatus ippsMul_8u16u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
     Ipp16u* pDst, int len);

IppStatus ippsMul_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
     Ipp32f* pDst, int len);

IppStatus ippsMul_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);

IppStatus ippsMul_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);

IppStatus ippsMul_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);

IppStatus ippsMul_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);

IppStatus ippsMul_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);

IppStatus ippsMul_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
     Ipp8u* pDst, int len, int scaleFactor);

IppStatus ippsMul_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
     Ipp16s* pDst, int len, int scaleFactor);

IppStatus ippsMul_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
     Ipp32s* pDst, int len, int scaleFactor);

IppStatus ippsMul_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
     Ipp16sc* pDst, int len, int scaleFactor);

IppStatus ippsMul_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2,
     Ipp32sc* pDst, int len, int scaleFactor);

IppStatus ippsMul_16u16s_Sfs(const Ipp16u* pSrc1, const Ipp16s* pSrc2,
     Ipp16s* pDst, int len, int scaleFactor);

IppStatus ippsMul_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
     Ipp32s* pDst, int len, int scaleFactor);

IppStatus ippsMul_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len,
     int scaleFactor);

IppStatus ippsMul_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst,
     int len, int scaleFactor);

IppStatus ippsMul_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int
     len, int scaleFactor);
```

```
IppStatus ippsMul_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int
      len, int scaleFactor);
IppStatus ippsMul_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int
      len, int scaleFactor);
```

Arguments

pSrc1, pSrc2	Pointers to the vectors whose elements are to be multiplied.
pDst	Pointer to the destination vector which stores the result of the multiplication *pSrc1[n] * pSrc2[n]*.
pSrc	Pointer to the vector whose elements are to be multiplied by the elements of *pSrcDst* in-place.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function `ippsMul` multiplies the elements of the vector *pSrc1* by the elements of the vector *pSrc2* and stores the result in *pDst*.

The in-place flavors of `ippsMul` multiply the elements of the vector *pSrc* by the elements of the vector *pSrcDst* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Value

`ippStsNoErr`	Indicates no error
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0

SubC

Subtracts a constant value from each element of a vector.

```
IppStatus ippsSubC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst,
     int len);
IppStatus ippsSubC_32fc(const Ipp32fc* pSrc, Ipp32fc val,
     Ipp32fc* pDst, int len)
IppStatus ippsSubC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst,
     int len);
IppStatus ippsSubC_64fc(const Ipp64fc* pSrc, Ipp64fc val,
     Ipp64fc* pDst, int len);
IppStatus ippsSubC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsSubC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsSubC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsSubC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsSubC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsSubC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
     int len, int scaleFactor);
IppStatus ippsSubC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val,
     Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsSubC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val,
     Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsSubC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val,
     Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsSubC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val,
     Ipp32sc* pDst, int len, int scaleFactor);
IppStatus ippsSubC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsSubC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsSubC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len,
     int scaleFactor);
```

```
IppStatus ippsSubC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsSubC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len,
     int scaleFactor);
```

Arguments

val	Scalar value used to decrement each element of the vector *pSrc* or *pSrcDst*.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector which stores the result of the subtraction *pSrc[n] - val*.
pSrcDst	Pointer to the vector whose elements are to be decreased by the value *val* and stored in *pSrcDst* for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsSubC subtracts a value *val* from each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of ippsSubC subtract a value *val* from each element of the vector *pSrcDst* and store the result in *pSrcDst*.

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

SubCRev

Subtracts each element of a vector from a constant value.

```
IppStatus ippsSubCRev_32f(const Ipp32f* pSrc, Ipp32f val,
     Ipp32f* pDst, int len);
IppStatus ippsSubCRev_64f(const Ipp64f* pSrc, Ipp64f val,
     Ipp64f* pDst, int len);
IppStatus ippsSubCRev_32fc(const Ipp32fc* pSrc, Ipp32fc val,
     Ipp32fc* pDst, int len);
IppStatus ippsSubCRev_64fc(const Ipp64fc* pSrc, Ipp64fc val,
     Ipp64fc* pDst, int len);
IppStatus ippsSubCRev_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsSubCRev_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsSubCRev_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsSubCRev_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsSubCRev_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
     int len, int scaleFactor);
IppStatus ippsSubCRev_16s_Sfs(const Ipp16s* pSrc, Ipp16s val,
     Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsSubCRev_32s_Sfs(const Ipp32s* pSrc, Ipp32s val,
     Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsSubCRev_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val,
     Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsSubCRev_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val,
     Ipp32sc* pDst, int len, int scaleFactor))
IppStatus ippsSubCRev_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst,
     int len, int scaleFactor);
IppStatus ippsSubCRev_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsSubCRev_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len,
     int scaleFactor);
```

```
IppStatus ippsSubCRev_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsSubCRev_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len,
     int scaleFactor);
```

Arguments

val	Scalar value from which vector elements are subtracted.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the vector whose elements are to be subtracted from the value *val* in case of the in-place operation. The destination vector which stores the result of the subtraction *val - pSrcDst[n]*.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsSubCRev subtracts each element of the vector *pSrc* from a value *val* and stores the result in *pDst*.

The in-place flavors of ippsSubCRev subtract each element of the vector *pSrcDst* from a value *val* and store the result in *pSrcDst*.

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Sub

Subtracts the elements of two vectors.

```
IppStatus ippsSub_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
     Ipp16s* pDst, int len);
IppStatus ippsSub_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
     Ipp32f* pDst, int len);
IppStatus ippsSub_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
     Ipp64f* pDst, int len);
IppStatus ippsSub_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
     Ipp32fc* pDst, int len);
IppStatus ippsSub_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
     Ipp64fc* pDst, int len);
IppStatus ippsSub_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
     Ipp32f* pDst, int len);
IppStatus ippsSub_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsSub_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsSub_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsSub_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsSub_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsSub_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
     Ipp8u* pDst, int len, int scaleFactor);
IppStatus ippsSub_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
     Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsSub_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
     Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsSub_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
     Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsSub_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2,
     Ipp32sc* pDst, int len, int scaleFactor);
IppStatus ippsSub_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst,
     int len, int scaleFactor);
```

```
IppStatus ippsSub_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst,
     int len, int scaleFactor);
IppStatus ippsSub_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst,
     int len, int scaleFactor);
IppStatus ippsSub_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst,
     int len, int scaleFactor);
IppStatus ippsSub_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst,
     int len, int scaleFactor)
```

Arguments

pSrc1	Pointer to the source vector whose elements are to be subtracted from *pSrc2*.
pSrc2	Pointer to the source vector whose elements are to be decreased by the elements of *pSrc1*.
pDst	Pointer to the destination vector which stores the result of the subtraction *pSrc2[n]* - *pSrc1[n]*.
pSrc	Pointer to the vector whose elements are to be subtracted from the elements of *pSrcDst* in-place.
pSrcDst	Pointer to the vector whose elements are to be decreased by the elements of *pSrc* and stored in *pSrcDst* for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsSub subtracts the elements of the vector *pSrc1* from the elements of the vector *pSrc2*, and stores the result in *pDst*.

The in-place flavors of ippsSub subtract the elements of the vector *pSrc* from the elements of a vector *pSrcDst* and store the result in *pSrcDst*.

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Value

ippStsNoErr	Indicates no error
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0

DivC

Divides each element of a vector by a constant value.

```
IppStatus ippsDivC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst,
     int len);
IppStatus ippsDivC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst,
     int len);
IppStatus ippsDivC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc*
     pDst, int len);
IppStatus ippsDivC_64fc(const Ipp64fc* pSrc, Ipp64fc val,
     Ipp64fc* pDst, int len);
IppStatus ippsDivC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsDivC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsDivC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsDivC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsDivC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
     int len, int ScaleFactor);
IppStatus ippsDivC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s*
     pDst, int len, int ScaleFactor);
IppStatus ippsDivC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val,
     Ipp16sc* pDst, int len, int ScaleFactor);
IppStatus ippsDivC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len,
     int ScaleFactor);
IppStatus ippsDivC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len,
     int ScaleFactor);
IppStatus ippsDivC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
      int ScaleFactor);
```

Arguments

val	Scalar value used to divide each element of the vector *pSrc* or *pSrcDst*.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector which stores the result of the division *pSrc[n] / val*.
pSrcDst	Pointer to the vector whose elements are divided by the value *val* and stored in *pSrcDst* for the in-place operation.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsDivC divides each element of the vector *pSrc* by a value *val* and stores the result in *pDst*.

The in-place flavors of ippsDivC divide each element of the vector *pSrcDst* by a value *val* and store the result in *pSrcDst*.

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZeroErr	Indicates an error when *val* is equal to 0.

DivCRev

Divides a constant value by each element of a vector.

```
IppStatus ippsDivCRev_16u(const Ipp16u* pSrc, Ipp16u val,
     Ipp16u* pDst, int len);
IppStatus ippsDivCRev_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
```

Arguments

val	Constant value that is used as a dividend in the operation.
pSrc	Pointer to the source vector whose elements are used as divisors.
pDst	Pointer to the destination vector which stores the result of the division *val / pSrc[n]*.
pSrcDst	Pointer to the vector whose elements are used both as divisors for the in-place operation and to store the resulting quotients.
len	Number of elements in the vector

Discussion

The function ippsDivCRev divides the constant value *val* by each element of the vector *pSrc* and stores the results in *pDst*.
The in-place flavors of ippsDivC divide the constant value *val* by each element of the vector *pSrcDst* and store the results in *pSrcDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZeroErr	Indicates an error when *val* is equal to 0.

Div

Divides the elements of two vectors.

```
IppStatus ippsDiv_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
     Ipp32f* pDst, int len);
IppStatus ippsDiv_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
     Ipp64f* pDst, int len);
IppStatus ippsDiv_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
     Ipp32fc* pDst, int len);
IppStatus ippsDiv_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
     Ipp64fc* pDst, int len);
IppStatus ippsDiv_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsDiv_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsDiv_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsDiv_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsDiv_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst,
     int len, int ScaleFactor);
IppStatus ippsDiv_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst,
     int len, int ScaleFactor);
IppStatus ippsDiv_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst,
     int len, int ScaleFactor);
IppStatus ippsDiv_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
     Ipp8u* pDst, int len, int scaleFactor);
IppStatus ippsDiv_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
     Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsDiv_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
     Ipp16sc* pDst, int len, int scaleFactor);
```

Arguments

pSrc1	Pointer to the vector whose elements are used as divisors for the elements of *pSrc2*.

pSrc2	Pointer to the vector whose elements are to be divided by the elements of *pSrc1*.
pDst	Pointer to the destination vector *pDst* which stores the result of the division *pSrc2[n]* / *pSrc1[n]*.
pSrc	Pointer to the source vector whose elements are used as divisors for the elements of *pSrcDst* in case of the in-place operation.
pSrcDst	Pointer to the vector whose elements are to be divided by the elements of *pSrc* and stored in *pSrcDst* for the in-place operation.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsDiv divides the elements of the vector *pSrc2* by the elements of the vector *pSrc1*, and stores the result in *pDst*.

The in-place flavors of ippsDiv divide the elements of the vector *pSrcDst* by the elements of the vector *pSrc* and store the result in *pSrcDst*.

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Example 5-3 shows that the usage of the scaling factor in the integer functions increases operation accuracy. Example 5-4 considers division by zero exceptions (x / 0, 0 / 0).

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

`ippStsDivByZero`	Indicates a warning for zero-valued divisor vector element. Operation execution is not aborted. The value of the destination vector element in the floating-point operations:
`NaN`	For zero-valued dividend vector element.
`+Inf`	For positive dividend vector element.
`-Inf`	For negative dividend vector element.

Example 5-3 Using the ippsDiv_16s_ISfs Function

```
IppStatus div16s( void ) {
      Ipp16s x[4] = { -3, 2, 0, 300 };
      Ipp16s y[4] = { -2, 2, 0, 0 };
      IppStatus st = ippsDiv_16s_ISfs( y, x, 4, -1 );
      printf_16s("div16s =", x, 4, st );
      return st;
}
Output:
    -- warning 6, Zero value(s) in the divisor of the function Div
    div16s =  3 2 0 32767
```

Example 5-4 Using the ippsDiv_32f_I Function

```
IppStatus div32f( void ) {
      Ipp32f x[4] = { -3, 2, 0, 300 };
      Ipp32f y[4] = { -2, 2, 0, 0 };
      IppStatus st = ippsDiv_32f_I( y, x, 4 );
      printf_32f( "div32f =", x, 4, st );
      return st;
}
Output:
    -- warning 6, Zero value(s) in the divisor of the function Div
    div32f =  1.500000 1.000000 1.#IND00 1.#INF00
```

Abs

Computes absolute values of vector elements.

```
IppStatus ippsAbs_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsAbs_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAbs_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAbs_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsAbs_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsAbs_64f_I(Ipp64f* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsAbs computes the absolute values of each element of the vector *pSrc* and stores the result in *pDst*.
The in-place flavors of ippsAbs compute the absolute values of each element of the vector *pSrcDst* and store the result in *pSrcDst*.

To compute the absolute values of complex data, use the function ippsMagnitude.

Example 5-5 shows how to call the function ippsAbs_32f_I.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-5 Using the ippsAbs Function

```
void abs32f(void) {
      Ipp32f x[4] = {-1, 1, 0, 0};
      x[3] *= (-1);
      ippsAbs_32f_I(x, 4);
      printf_32f("abs =", x, 4, ippStsNoErr);
}
Output:
    abs =  1.000000 1.000000 0.000000 0.000000
```

Sqr

Computes a square of each element of a vector.

```
IppStatus ippsSqr_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSqr_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSqr_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSqr_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSqr_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSqr_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsSqr_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsSqr_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsSqr_8u_Sfs(const Ipp8u* pSrc, Ipp8u* pDst, int len,
     int scaleFactor);
IppStatus ippsSqr_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
     int scaleFactor);
IppStatus ippsSqr_16u_Sfs(const Ipp16u* pSrc, Ipp16u* pDst, int len,
     int scaleFactor);
```

```
IppStatus ippsSqr_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
     int scaleFactor);
IppStatus ippsSqr_8u_ISfs(Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16u_ISfs(Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16sc_ISfs(Ipp16sc* pSrcDst, int len, int
     scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsSqr computes the square of each element of the vector *pSrc*, and stores the result in *pDst*. The computation is performed as follows:

$pDst[n] = pSrc[n]^2$

The in-place flavors of ippsSqr compute the square of each element of the vector *pSrcDst* and store the result in *pSrcDst*. The computation is performed as follows:

$pSrcDst[n] = pSrcDst[n]^2$

When computing the square of an integer number, the output result can exceed the data range and become saturated. To get a precise result, use the scale factor. Example 5-6 shows how to get the value of 200^2. Without scaling this result is clipped to 32767. Scaling retains the output data range but results in the precision loss in low-order bits.

Return Value

ippStsNoErr	Indicates no error.

`ippStsNullPtrErr`	Indicates an error when the *`pSrc`*, *`pDst`*, *`or pSrcDst`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.

Example 5-6 Using the ippsSqr Function

```
IppStatus sqr(void) {
      Ipp16s x[4] = {-3, 2, 30, 200};
      IppStatus st = ippsSqr_16s_ISfs(x, 4, 1);
      printf_16s("sqr =", x, 4, st);
      return st;
}
Output:
    sqr =  4 2 450 20000
```

Sqrt

Computes a square root of each element of a vector.

```
IppStatus ippsSqrt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSqrt_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSqrt_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSqrt_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSqrt_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSqrt_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsSqrt_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsSqrt_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsSqrt_8u_Sfs(const Ipp8u* pSrc, Ipp8u* pDst, int len,
     int scaleFactor);
```

```
IppStatus ippsSqrt_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, int scaleFactor);
IppStatus ippsSqrt_16u_Sfs(const Ipp16u* pSrc, Ipp16u* pDst, int len,
     int scaleFactor);
IppStatus ippsSqrt_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, int scaleFactor);
IppStatus ippsSqrt_64s_Sfs(const Ipp64s* pSrc, Ipp64s* pDst, int len,
     int scaleFactor);
IppStatus ippsSqrt_8u_ISfs(Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16u_ISfs(Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16sc_ISfs(Ipp16sc* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_64s_ISfs(Ipp64s* pSrcDst, int len, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsSqrt computes the square root of each element of the vector *pSrc*, and stores the result in *pDst*. The computation is performed as follows:

$$pDst[n] = \sqrt{pSrc[n]}$$

The in-place flavors of ippsSqrt compute the square root of each element of the vector *pSrcDst* and store the result in *pSrcDst*. The computation is performed as follows:

$$pSrcDst[n] = \sqrt{pSrcDst[n]}$$

The square root of complex vector elements is computed as follows:

$$\sqrt{a + j \cdot b} = \sqrt{\frac{\sqrt{a^2 + b^2} + a}{2}} + j \cdot sign(b) \cdot \sqrt{\frac{\sqrt{a^2 + b^2} - a}{2}}$$

Application Notes

If the function `ippsSqrt` encounters a negative value in the input, it returns a warning status. Operation execution is not aborted. To increase precision of an integer output, use the scale factor.

Example 5-7 shows how to call the function `ippsSqrt_16s_ISfs`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrc`*, *`pDst`*, *`or pSrcDst`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`ippStsSqrtNegArg`	Indicates a warning. Operation execution is not aborted. The value of the destination vector element is:
	`NaN` For negative input vector element in floating-point operations.
	0 For negative input vector element in integer operations.

Example 5-7 Using the ippsSqrt Function

```
IppStatus sqrt(void) {
      Ipp16s x[4] = {-3, 2, 30, 300};
      IppStatus st = ippsSqrt_16s_ISfs(x, 4, -1);
      printf_16s("sqrt =", x, 4, st);
      return st;
}
Output:
    -- warning 3, Negative value(s) in the argument of the function Sqrt
    sqrt = 0 3 11 35
```

Cubrt

Computes cube root of each element of a vector.

```
IppStatus ippsCubrt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCubrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst,
     int len, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function `ippsCubrt` computes cube root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

The computation is performed as follows:

$pDst[n] = \sqrt[3]{pSrc[n]}, 0 \le n < len.$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Exp

Computes e *to the power of each element of a vector.*

```
IppStatus ippsExp_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsExp_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsExp_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
     int scaleFactor);
IppStatus ippsExp_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len,
     int scaleFactor);
IppStatus ippsExp_64s_Sfs(const Ipp64s* pSrc, Ipp64s* pDst, int len,
     int scaleFactor);
IppStatus ippsExp_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsExp_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsExp_64s_ISfs(Ipp64s* pSrcDst, int len, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.

pSrcDst	Pointer to the source and destination vector *pSrcDst* for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsExp computes the exponential function of each element of the vector *pSrc*, and stores the result in *pDst*.

The computation is performed as follows:

$pDst[n] = e^{pSrc[n]}$

The in-place flavors of ippsExp compute the exponential function of each element of the vector *pSrcDst* and store the result in *pSrcDst*.

The computation is performed as follows:

$pSrcDst[n] = e^{pSrcDst[n]}$

When computing the exponent of an integer number, the output result can exceed the data range and become saturated. The scaling retains the output data range but results in precision loss in low-order bits. The function ippsExp_32f64f computes the output result in a higher precision data range.

Example 5-8 shows how to call the function ippsExp_16s_ISfs. Example 5-9 shows how to call the function ippsExp_64f_I.

Application Notes

For the functions ippsExp and ippsLn the result is rounded to the nearest integer after scaling.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-8 Using the ippsExp_16s_ISfs Function

```
IppStatus exp16s(void) {
      Ipp16s x[4] = {-1, 2, 30, 0};
      IppStatus st = ippsExp_16s_ISfs(x, 4, -1);
      printf_16s("exp16s =", x, 4, st);
      return st;
}
Output:
    exp16s =  1 15 32767 2
```

Example 5-9 Using the ippsExp_64f_I Function

```
IppStatus exp64f(void) {
      Ipp64f x[4] = {-1, 2, 1, log(1.234567)};
      IppStatus st = ippsExp_64f_I(x, 4);
      printf_64f("exp64f =", x, 4, st);
      return st;
}
Output:
    exp64f =  0.367879 7.389056 2.718282 1.234567
```

Ln

Computes the natural logarithm of each element of a vector.

```
IppStatus ippsLn_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLn_64f32f(const Ipp64f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsLn_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsLn_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsLn_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
     int scaleFactor);
IppStatus ippsLn_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len,
     int scaleFactor))
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsLn computes the natural logarithm of each element of the vector *pSrc* and stores the result in *pDst* as given by

$pDst[n] = \log_e(pSrc[n])$

The in-place flavors of ippsLn compute the natural logarithm of each element of the vector *pSrcDst* and store the result in *pSrcDst* as given by

pSrcDst[n] = $\log_e$ (*pSrcDst[n]*)

Example 5-10 shows how to call the function ippsLn_32f_I.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsLnZeroArg	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is -Inf.
ippStsLnNegArg	Indicates a warning for negative input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is NaN.

Example 5-10 Using the ippsLn Function

```
IppStatus ln32f(void) {
      Ipp32f x[4] = {-1, (float)IPP_E, 0, (float)(exp(1.234567))};
      IppStatus st = ippsLn_32f_I(x, 4);
      printf_32f("Ln =", x, 4, st);
      return st;
}
Output:
    -- warning 8, Negative value(s) of argument in the Ln function
    Ln = -1.#IND00 1.000000 -1.#INF00 1.234567
```

SumLn

Sums natural logarithms of each element of a vector.

```
IppStatus ippsSumLn_32f(const Ipp32f* pSrc, int len, Ipp32f* pSum);
IppStatus ippsSumLn_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSumLn_32f64f(const Ipp32f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSumLn_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pSum);
```

Arguments

pSrc	Pointer to the source vector.
pSum	Pointer to the output result.
len	Number of elements in the vector.

Discussion

The function ippsSumLn computes the sum of natural logarithms of each element of the vector *pSrc* and stores the result value in *pSum*. The summation is given by:

$$sum = \sum_{n=0}^{len-1} \ln(pSrc[n])$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pSum* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsLnZeroArg	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to -Inf.

ippStsLnNegArg	Indicates a warning for negative input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to NaN.

Arctan

Computes the inverse tangent of each element of a vector.

```
IppStatus ippsArctan_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsArctan_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsArctan_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsArctan_64f_I(Ipp64f* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector *pSrcDst* for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsAtan computes the inverse tangent of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

The computation is performed as follows:

pDst[n] = arctan(*pSrc*[n]), $0 \le n < len$.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Normalize

Normalizes elements of a real or complex vector using offset and division operations.

```
IppStatus ippsNormalize_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
     Ipp32f vsub, Ipp32f vdiv);
IppStatus ippsNormalize_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
     Ipp64f vsub, Ipp64f vdiv);
IppStatus ippsNormalize_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len, Ipp32fc vsub, Ipp32f vdiv);
IppStatus ippsNormalize_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len, Ipp64fc vsub, Ipp64f vdiv);
IppStatus ippsNormalize_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, Ipp16s vsub, int vdiv, int scaleFactor);
IppStatus ippsNormalize_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, Ipp16sc vsub, int vdiv, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
vsub	Subtrahend value.
vdiv	Denominator value.
pDst	Pointer to the vector which stores the normalized elements.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsNormalize subtracts *vsub* from elements of the input vector *pSrc*, divides the differences by *vdiv*, and stores the result in *pDst*. The computation is performed as follows:

$$pDst[n] = \frac{pSrc[n] - vsub}{vdiv}$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZeroErr	Indicates an error when *vdiv* is equal to 0 or less than the minimum floating-point positive number.

Conversion Functions

The functions described in this section perform the following conversion operations for vectors:

- Extracting components from and constructing a complex vector
- Computing the complex conjugates of vectors
- Floating-point to integer and integer to floating point conversions
- Cartesian to polar and polar to Cartesian coordinate conversion

This section also describes the IPP functions that extract real and imaginary components from a complex vector or construct a complex vector using its real and imaginary components.

The functions ippsReal and ippsImag return the real and imaginary parts of a complex vector in a separate vector, respectively.
The function ippsRealToCplx constructs a complex vector from real and imaginary components stored in two respective vectors.
The function ippsCplxToReal returns the real and imaginary parts of a complex vector in two respective vectors.

The function ippsMagnitude computes the magnitude of a complex vector elements.

Convert

Converts the integer data of a vector to floating-point data or vice-versa, and stores the results in a second vector.

```
IppStatus ippsConvert_8s16s(const Ipp8s* pSrc,Ipp16s* pDst,int len);
IppStatus ippsConvert_8s32f(const Ipp8s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16u32f(const Ipp16u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32s32f(const Ipp32s* pSrc, Ipp32f* pDst,int len);
IppStatus ippsConvert_32s64f(const Ipp32s* pSrc, Ipp64f* pDst, int len);
IppStatus ippsConvert_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsConvert_64f32f(const Ipp64f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16s32f_Sfs(const Ipp16s* pSrc, Ipp32f* pDst,
     int len, int scaleFactor);
IppStatus ippsConvert_16s64f_Sfs(const Ipp16s* pSrc, Ipp64f* pDst,
     int len, int scaleFactor);
IppStatus ippsConvert_32s32f_Sfs(const Ipp32s* pSrc, Ipp32f* pDst,
     int len, int scaleFactor);
IppStatus ippsConvert_32s64f_Sfs(const Ipp32s* pSrc, Ipp64f* pDst,
     int len, int scaleFactor);
IppStatus ippsConvert_32f8s_Sfs(const Ipp32f* pSrc, Ipp8s* pDst,
     int len, IppRoundMode rndmode, int scaleFactor);
IppStatus ippsConvert_32f8u_Sfs(const Ipp32f* pSrc, Ipp8u* pDst,
     int len, IppRoundMode rndmode, int scaleFactor);
IppStatus ippsConvert_32f16s_Sfs(const Ipp32f* pSrc, Ipp16s* pDst,
     int len, IppRoundMode rndmode, int scaleFactor);
IppStatus ippsConvert_32f16u_Sfs(const Ipp32f* pSrc, Ipp16u* pDst,
     int len, IppRoundMode rndmode, int scaleFactor);
IppStatus ippsConvert_32f32s_Sfs(const Ipp32f* pSrc, Ipp32s* pDst,
     int len, IppRoundMode rndmode, int scaleFactor);
```

```
IppStatus ippsConvert_64f32s_Sfs(const Ipp64f* pSrc, Ipp32s* pDst,
     int len, IppRoundMode rndmode, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.	
pDst	Pointer to the destination vector.	
rndmode	Rounding mode which can be *ippRndZero* or *ippRndNear*:	
	ippRndZero	Specifies that floating-point values must be truncated toward zero.
	ippRndNear	Specifies that floating-point values must be rounded to the nearest integer.
len	Number of elements in the vector.	
scaleFactor	Refer to Integer Scaling.	

Discussion

The function `ippsConvert` converts the integer data in the vector *pSrc* to floating-point data and stores the results in *pDst*.

The function `ippsConvert` with the `Sfs` suffix converts floating-point data in the vector *pSrc* into integer data and stores the results in *pDst*. The converted result is saturated if exceeds the output data range.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Conj

Stores the complex conjugate values of a vector in a second vector or in-place.

```
IppStatus ippsConj_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsConj_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsConj_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsConj_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsConj_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsConj_32fc_I(Ipp32fc* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsConj stores in *pDst* the element-wise conjugation of the complex vector *pSrc*. The element-wise conjugation of the vector is defined as follows:

pDst[n].re = pSrc[n].re

pDst[n].im = - pSrc[n].im

The in-place flavors of ippsConj store in *pSrcDst* the element-wise conjugation of the complex vector *pSrcDst*.

The element-wise conjugation of the vector is defined as follows:

pSrcDst[n].re = pSrcDst[n].re

pSrcDst[n].im = - pSrcDst[n].im

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

ConjFlip

Computes the complex conjugate of a vector and stores the result in reverse order.

```
IppStatus ippsConjFlip_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len);
IppStatus ippsConjFlip_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len);
IppStatus ippsConjFlip_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len);
```

Arguments

pSrc	Pointer to the source vector *pSrc*.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsConjFlip computes the conjugate of the vector *pSrc* and stores the result, in reverse order, in *pDst*.

The complex conjugate, stored in reverse order, is defined as follows:

pDst[n] = conj(pSrc[len - n - 1])

The in-place flavors of ippsConjFlip compute the conjugate of the vector *pSrcDst* and store the result, in reverse order, in *pSrcDst*.

The complex conjugate, stored in reverse order, is defined as follows:

pSrcDst[n] = conj(*pSrcDst[len* - *n* - 1*]*)

Note that if *pSrc* and *pDst* overlap in memory, the function returns unpredictable results.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Magnitude

Computes the magnitudes of the elements of a complex vector.

```
IppStatus ippsMagnitude_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
      Ipp32f* pDst, int len);
IppStatus ippsMagnitude_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
      Ipp64f* pDst, int len);
IppStatus ippsMagnitude_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsMagnitude_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsMagnitude_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst,
      int len);
IppStatus ippsMagnitude_16s_Sfs(const Ipp16s* pSrcRe,
      const Ipp16s* pSrcIm, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsMagnitude_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst,
      int len, int scaleFactor);
IppStatus ippsMagnitude_32sc_Sfs(const Ipp32sc* pSrc,Ipp32s* pDst,
      int len,int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pSrcRe	Pointer to the vector with the real parts of complex elements.
pSrcIm	Pointer to the vector with the imaginary parts of complex elements.
pDst	Pointer to the destination vector.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The "complex" function ippsMagnitude computes the element-wise magnitude of the complex vector *pSrc* and stores the result in *pDst*. The element-wise magnitude is defined by the formula:

$$magn[n] = \sqrt{pSrc[n].re^2 + pSrc[n].im^2}$$

The "real" function ippsMagnitude computes the element-wise magnitude of the complex vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, and stores the result in *pDst*. The element-wise magnitude is defined by the formula:

$$magn[n] = \sqrt{pSrcRe[n]^2 + pSrcIm[n]^2}$$

Example 5-11 verifies the identity $\sin^2 x + \cos^2 x = 1$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-11 Using the ippsMagnitude Function

```
void magn(void) {
      Ipp64f x[6], magn[4];
      int n;
      for (n = 0; n<6; ++n) x[n] = sin(IPP_2PI * n / 8);
      ippsMagnitude_64f(x, x+2, magn, 4);
      printf_64f("magn =", magn, 4, ippStsNoErr);
}
Output:
    magn =  1.000000 1.000000 1.000000 1.000000
Matlab* Analog:
    >> n = 0:9; x = sin(2*pi*n/8); z = [x(1:8)+j*x(3:10)]; abs(z(1:4))
```

Phase

Computes the phase angles of elements of a complex vector.

```
IppStatus ippsPhase_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsPhase_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int
      len, int scaleFactor);
IppStatus ippsPhase_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
      Ipp64f* pDst, int len);
IppStatus ippsPhase_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
      Ipp32f* pDst, int len);
IppStatus ippsPhase_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
      Ipp32f* pDst, int len);
```

```
IppStatus ippsPhase_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
     Ipp16s* pDst, int len, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pSrcRe	Pointer to the source vector which stores the real components.
pSrcIm	Pointer to the source vector which stores the imaginary components.
pDst	Pointer to the vector which stores the phase (angle) components of the elements in radians. Phase values are in the range (-π,π].
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsPhase returns the phase angles of elements of the complex input vector *pSrc*, or the complex input vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, and stores the result in the vector *pDst*. Phase values are returned in radians and are in the range (-π, π].

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

PowerSpectr

Computes the power spectrum of a complex vector.

```
IppStatus ippsPowerSpectr_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsPowerSpectr_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPowerSpectr_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst,int
      len, int scaleFactor);
IppStatus ippsPowerSpectr_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst,
      int len);
IppStatus ippsPowerSpectr_64f(const Ipp64f* pSrcRe, const Ipp64f*
      pSrcIm, Ipp64f* pDst, int len);
IppStatus ippsPowerSpectr_32f(const Ipp32f* pSrcRe, const Ipp32f*
      pSrcIm, Ipp32f* pDst, int len);
IppStatus ippsPowerSpectr_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s*
      pSrcIm, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsPowerSpectr_16s32f(const Ipp16s* pSrcRe, const Ipp16s*
      pSrcIm, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pSrcRe	Pointer to the source vector which stores the real components.
pSrcIm	Pointer to the source vector which stores the imaginary components.
pDst	Pointer to the vector which stores the spectrum components of the elements.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsPowerSpectr returns the power spectrum of the complex input vector *pSrc*, or the complex input vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, and stores the results in the vector *pDst*. The power spectrum elements are squares of the magnitudes of the complex input vector elements:

pDst[n] = (*pSrc[n].re*)^2 + (*pSrc[n].im*)^2 , or
pDst[n] = (*pSrcRe[n]*)^2 + (*pSrcIm[n]*)^2 .

To compute magnitudes, use the function ippsMagnitude on page 5-53.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Real

Returns the real part of a complex vector in a second vector.

```
IppStatus ippsReal_16sc(const Ipp16sc* pSrc, Ipp16s* pDstRe, int len);
IppStatus ippsReal_32fc(const Ipp32fc* pSrc, Ipp32f* pDstRe, int len);
IppStatus ippsReal_64fc(const Ipp64fc* pSrc, Ipp64f* pDstRe, int len);
```

Arguments

pSrc	Pointer to the complex source vector.
pDstRe	Pointer to the destination vector with real parts.
len	Number of elements in the vector.

Discussion

The function ippsReal returns the real part of the complex vector *pSrc* in the vector *pDstRe*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDstRe* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Imag

Returns the imaginary part of a complex vector in a second vector.

```
IppStatus ippsImag_16sc(const Ipp16sc* pSrc, Ipp16s* pDstIm, int len);
IppStatus ippsImag_32fc(const Ipp32fc* pSrc, Ipp32f* pDstIm, int len);
IppStatus ippsImag_64fc(const Ipp64fc* pSrc, Ipp64f* pDstIm, int len);
```

Arguments

pSrc	Pointer to the complex source vector.
pDstIm	Pointer to the destination vector with imaginary parts.
len	Number of elements in the vector.

Discussion

The function ippsImag returns the imaginary part of a complex vector *pSrc* in the vector *pDstIm*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDstIm* or *pSrc* pointer is NULL.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

RealToCplx

Returns a complex vector constructed from the real and imaginary parts of two real vectors.

```
IppStatus ippsRealToCplx_16s(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
     Ipp16sc* pDst, int len);
IppStatus ippsRealToCplx_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
     Ipp32fc* pDst, int len);
IppStatus ippsRealToCplx_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
     Ipp64fc* pDst, int len);
```

Arguments

pSrcRe	Pointer to the vector with real parts of complex elements.
pSrcIm	Pointer to the vector with imaginary parts of complex elements.
pDst	Pointer to the destination vector.
len	Number of elements in the vector.

Discussion

The function ippsRealToCplx returns a complex vector *pDst* constructed from the real and imaginary parts of the input vectors *pSrcRe* and *pSrcIm*.

If *pSrcRe* is NULL, the real component of the vector is set to zero.

If *pSrcIm* is NULL, the imaginary component of the vector is set to zero.

Note that both pointers can not be NULL.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pDst* pointer is NULL. The pointer *pSrcRe* or *pSrcIm* can be NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

CplxToReal

Returns the real and imaginary parts of a complex vector in two respective vectors.

```
IppStatus ippsCplxToReal_16sc(const Ipp16sc* pSrc, Ipp16s* pDstRe,
     Ipp16s* pDstIm, int len);
IppStatus ippsCplxToReal_32fc(const Ipp32fc* pSrc, Ipp32f* pDstRe,
     Ipp32f* pDstIm, int len);
IppStatus ippsCplxToReal_64fc(const Ipp64fc* pSrc, Ipp64f* pDstRe,
     Ipp64f* pDstIm, int len);
```

Arguments

pSrc	Pointer to the complex vector *pSrc*.
pDstRe	Pointer to the output vector with real parts.
pDstIm	Pointer to the output vector with imaginary parts.
len	Number of elements in the vector.

Discussion

The function ippsCplxToReal returns the real and imaginary parts of a complex vector *pSrc* in two vectors *pDstRe* and *pDstIm*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the data vector pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Threshold

Performs the threshold operation on the elements of a vector by limiting the element values by `level`.

```
IppStatus ippsThreshold_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
      Ipp16s level, IppCmpOp relOp);
IppStatus ippsThreshold_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
      Ipp32f level, IppCmpOp relOp);
IppStatus ippsThreshold_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
      Ipp64f level, IppCmpOp relOp);
IppStatus ippsThreshold_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
       Ipp32f level, IppCmpOp relOp);
IppStatus ippsThreshold_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
       Ipp64f level, IppCmpOp relOp);
IppStatus ippsThreshold_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
       Ipp16s level, IppCmpOp relOp);
IppStatus ippsThreshold_16s_I(Ipp16s* pSrcDst, int len,
      Ipp16s level, IppCmpOp relOp);
IppStatus ippsThreshold_32f_I(Ipp32f* pSrcDst, int len,
      Ipp32f level, IppCmpOp relOp);
IppStatus ippsThreshold_64f_I(Ipp64f* pSrcDst, int len,
      Ipp64f level, IppCmpOp relOp);
IppStatus ippsThreshold_32fc_I(Ipp32fc* pSrcDst, int len,
      Ipp32f level, IppCmpOp relOp);
IppStatus ippsThreshold_64fc_I(Ipp64fc* pSrcDst, int len,
      Ipp64f level, IppCmpOp relOp);
IppStatus ippsThreshold_16sc_I(Ipp16sc* pSrcDst, int len,
      Ipp16s level, IppCmpOp relOp);
```

Arguments

`pSrc`	Pointer to the source vector.
`pDst`	Pointer to the destination vector.

pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real. For complex versions, it must be positive and represent magnitude.
relOp	Values of this argument specify which relational operator to use and whether *level* is an upper or lower bound for the input. The *relOp* must have one of the following values:
ippCmpLess	Specifies the “less than” operator and *level* is a lower bound.
ippCmpGreater	Specifies the “greater than” operator and *level* is an upper bound.

Discussion

The function ippsThreshold performs the threshold operation on the vector *pSrc* by limiting each element by the threshold value *level*.

The in-place flavors of ippsThreshold perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value *level*.

The *relOp* argument specifies which relational operator to use: “greater than” or “less than,” and determines whether *level* is an upper or lower bound for the input, respectively. The formula for ippsThreshold called with the *ippCmpLess* flag is:

$$pDst[n] = \begin{cases} level, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function ippsThreshold, the *level* argument is always real. The formula for complex ippsThreshold called with the *ippCmpLess* flag is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

Application Notes

For all complex versions, *level* must be positive and represents a magnitude. The magnitude of the input is limited, but the phase remains unchanged. Zero-valued input is assumed to have zero phase.

A special rule is applied to the integer complex versions of the function ippsThreshold. In general, the resulting point coordinates at the complex plane are not integer. The function rounds them off to integer in such a way that the threshold operation is not performed. Thus, for the "less than" operation (with the *ippCmpLess* flag) the coordinates are rounded to the infinity (+Inf for positive coordinates, and -Inf for negative), and for the "greater than" operation (with the *ippCmpGreater* flag) the coordinates are rounded to 0. See Example 5-13 for more information about using the "complex" function ippsThreshold_16sc_I.

Example 5-12 shows how to use the "real" function ippsTreshold_16s_I.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsThreshNegLevelErr	Indicates an error when *level* for the complex version is negative.

Example 5-12 Using the Real ippsThreshold Function

```
IppStatus threshold( void ) {
     Ipp16s x[4] = { -1, 0, 2, 3 };
     IppStatus st = ippsThreshold_16s_I( x, 4, 2, ippCmpLess );
     printf_16s("threshold result =", x, 4, st );
     return st;
}
Output:
    threshold result =  2 2 2 3
```

Example 5-13 Using the Complex ippsThreshold Function

```
IppStatus cmplx_threshold(void) {
     Ipp16sc x[4] = {{2,3}, {3,3}, {4,3}, {4,2}};
     /// level is near to the point {2,3} = 3.6
     /// the point {2,3} is to be replaced
     /// the computed coordinates are {2.2188,3.3282}
     /// the point used is {3,4};
     /// notice that it is the point with the phase,
     /// nearest to the source
     IppStatus st = ippsThreshold_16sc_I(x, 4, 4, ippCmpLess);
     printf_16sc("complex threshold result =", x, 4, st);
     return st;
}
Output:
    complex threshold result = {3, 4} {3, 3} {4, 3} {4, 2}
```

Threshold_LT, Threshold_GT

Performs the threshold operation on the elements of a vector by limiting the element values by `level`.

```
IppStatus ippsThreshold_LT_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, Ipp16s level);
IppStatus ippsThreshold_LT_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len, Ipp32f level);
IppStatus ippsThreshold_LT_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len, Ipp64f level);
IppStatus ippsThreshold_LT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len, Ipp32f level);
```

```
IppStatus ippsThreshold_LT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len, Ipp64f level);
IppStatus ippsThreshold_LT_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, Ipp16s level);
IppStatus ippsThreshold_LT_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_LT_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, Ipp16s level);
IppStatus ippsThreshold_GT_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len, Ipp32f level);
IppStatus ippsThreshold_GT_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len, Ipp64f level);
IppStatus ippsThreshold_GT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len, Ipp32f level);
IppStatus ippsThreshold_GT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len, Ipp64f level);
IppStatus ippsThreshold_GT_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, Ipp16s level);
IppStatus ippsThreshold_GT_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.

pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real. For complex versions, it must be positive and represent magnitude.

Discussion

These functions perform the threshold operation on the vector *pSrc* by limiting each element by the threshold value *level*.

The in-place flavors perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value *level*.

ippsThreshold_LT. The ippsThreshold_LT function performs the operation “less than”, and *level* is a lower bound for the input. The formula for ippsThreshold_LT is the following:

$$pDst[n] = \begin{cases} level, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function ippsThreshold_LT, the *level* argument is always real.

The formula for complex ippsThreshold_LT is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

ippsThreshold_GT. The function ippsThreshold_GT performs the operation “greater than” and *level* is an upper bound for the input.

The formula for ippsThreshold_GT is:

$$pDst[n] = \begin{cases} level, & pSrc[n] > level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function ippsThreshold_GT, the *level* argument is always real.

The formula for complex ippsThreshold_GT is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) > level \\ pSrc[n], & otherwise \end{cases}$$

Application Notes

For all complex versions, *level* must be positive and represents a magnitude. The magnitude of the input is limited, but the phase remains unchanged. Zero-valued input is assumed to have zero phase.

A special rule is applied to the integer complex versions of the threshold functions. In general, the resulting point coordinates at the complex plane are not integer. The function rounds them off to integer in such a way that the threshold operation is not performed. Thus, for the "less than" operation (the ippsThreshold_LT function) the coordinates are rounded to the infinity (+Inf for positive coordinates, and -Inf for negative), and for the "greater than" operation (the ippsThreshold_GT function) the coordinates are rounded to 0.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0
ippStsThreshNegLevelErr	Indicates an error when *level* for the complex version is negative.

Threshold_LTVal, Threshold_GTVal

Performs the threshold operation on the elements of a vector by limiting the element values by `level`.

```
IppStatus ippsThreshold_LTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_LTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len, Ipp32f level, Ipp32f value);
IppStatus ippsThreshold_LTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len, Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_LTVal_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, Ipp16s level, Ipp16sc value);
IppStatus ippsThreshold_LTVal_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len, Ipp32f level, Ipp32fc value);
IppStatus ippsThreshold_LTVal_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len, Ipp64f level, Ipp64fc value);
IppStatus ippsThreshold_LTVal_16s_I(Ipp16s* pSrcDst, int len,
     Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_LTVal_32f_I(Ipp32f* pSrcDst, int len,
     Ipp32f level, Ipp32f value);
IppStatus ippsThreshold_LTVal_64f_I(Ipp64f* pSrcDst, int len,
     Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_LTVal_16sc_I(Ipp16sc* pSrcDst, int len,
     Ipp16s level, Ipp16sc value);
IppStatus ippsThreshold_LTVal_32fc_I(Ipp32fc* pSrcDst, int len,
     Ipp32f level, Ipp32fc value);
IppStatus ippsThreshold_LTVal_64fc_I(Ipp64fc* pSrcDst, int len,
     Ipp64f level, Ipp64fc value);
IppStatus ippsThreshold_GTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_GTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int len, Ipp32f level, Ipp32f value);
```

```
IppStatus ippsThreshold_GTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len, Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_GTVal_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, Ipp16s level, Ipp16sc value);
IppStatus ippsThreshold_GTVal_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len, Ipp32f level, Ipp32fc value);
IppStatus ippsThreshold_GTVal_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len, Ipp64f level, Ipp64fc value);
IppStatus ippsThreshold_GTVal_16s_I(Ipp16s* pSrcDst, int len,
     Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_GTVal_32f_I(Ipp32f* pSrcDst, int len,
     Ipp32f level, Ipp32f value);
IppStatus ippsThreshold_GTVal_64f_I(Ipp64f* pSrcDst, int len,
     Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_GTVal_16sc_I(Ipp16sc* pSrcDst, int len,
     Ipp16s level, Ipp16sc value);
IppStatus ippsThreshold_GTVal_32fc_I(Ipp32fc* pSrcDst, int len,
     Ipp32f level, Ipp32fc value);
IppStatus ippsThreshold_GTVal_64fc_I(Ipp64fc* pSrcDst, int len,
     Ipp64f level, Ipp64fc value);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real. For complex versions, it must be positive and represent magnitude.
value	Value to be assigned to vector elements which are “less than” or “greater than” *level*.

Discussion

These functions perform the threshold operation on the vector *pSrc* by limiting each element by the threshold value *level*.

The in-place flavors of the function perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value *level*.

ippsThreshold_LTVal. The ippsThreshold_LTVal function performs the operation "less than" and *level* is a lower bound for the input. The vector elements less than *level* are set to *value*.

The formula for ippsThreshold_LTVal is:

$$pDst[n] = \begin{cases} value, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function ippsThreshold_LTVal, the *level* argument is always real.

The formula for complex ippsThreshold_LTVal is:

$$pDst[n] = \begin{cases} value, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

ippsThreshold_GTVal. The ippsThreshold_GTVal function performs the operation "greater than" and *level* is an upper bound for the input. The vector elements greater than *level* are set to *value*.

The formula for ippsThreshold_GtVal is:

$$pDst[n] = \begin{cases} value, & pSrc[n] > level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function ippsThreshold_GTVal, the *level* argument is always real.

The formula for complex ippsThreshold_GTVal is:

$$pDst[n] = \begin{cases} value, & abs(pSrc[n]) > level \\ pSrc[n], & otherwise \end{cases}$$

Application Notes

For all complex versions, *level* must be positive and represents a magnitude.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsThreshNegLevelErr	Indicates an error when *level* for the complex version is negative.

Threshold_LTInv

Computes the inverse of vector elements after limiting their magnitudes by the given lower bound.

```
IppStatus ippsThreshold_LTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len, Ipp32f level);
IppStatus ippsThreshold_LTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len, Ipp64f level);
IppStatus ippsThreshold_LTInv_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len, Ipp32f level);
IppStatus ippsThreshold_LTInv_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len, Ipp64f level);
IppStatus ippsThreshold_LTInv_32f_I(Ipp32f* pSrcDst, int len,
     Ipp32f level);
IppStatus ippsThreshold_LTInv_64f_I(Ipp64f* pSrcDst, int len,
     Ipp64f level);
IppStatus ippsThreshold_LTInv_32fc_I(Ipp32fc* pSrcDst, int len,
     Ipp32f level);
IppStatus ippsThreshold_LTInv_64fc_I(Ipp64fc* pSrcDst, int len,
     Ipp64f level);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real and positive.

Discussion

The function ippsThreshold_LTInv computes the inverse of elements of the vector *pSrc* and stores the result in *pDst*. The computation occurs after first limiting the magnitude of each element by the threshold value *level*.

The in-place flavors of ippsThreshold_LTInv compute the inverse of elements of the vector *pSrcDst* and store the result in *pSrcDst*. The computation occurs after first limiting the magnitude of each element by the threshold value *level*.

The threshold operation is performed to avoid division by zero. Since *level* represents a magnitude, it is always real and must be positive. The formula for ippsThreshold_LTInv is the following:

$$pDst[n] = \begin{cases} \frac{1}{level}, & abs(pSrc[n]) = 0 \\ \frac{abs(pSrc[n])}{pSrc[n] \cdot level}, & 0 < abs(pSrc[n]) < level \\ \frac{1}{pSrc[n]}, & otherwise \end{cases}$$

If the function encounters zero-valued vector elements and *level* is also 0, the output value is set to Inf (infinity), but operation execution is not aborted:

$$pDst[n] = \begin{cases} Inf, & pSrc[n] = 0 \\ \frac{1}{pSrc[n]}, & otherwise \end{cases}$$

Example 5-14 shows how to use the function ippsThreshold_LTInv_32f_I.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsThreshNegLevelErr	Indicates an error when *level* is negative.
ippStsInvZero	Indicates a warning when *level* and a vector element are equal to 0. Operation execution is not aborted. The value of the destination vector element is Inf.

Example 5-14 Using the ippsThreshold_LTInv Function

```
IppStatus invThreshold(void) {
     Ipp32f x[4] = {-1, 0, 2, 3};
     IppStatus st = ippsThreshold_LTInv_32f_I(x, 4, 0);
     printf_32f("inv threshold =", x, 4, st);
     return st;
}
Output:
   -- warning 4, INF result. Zero value met by invThreshold with zero level
    inv threshold = -1.000000 1.#INF00 0.500000 0.333333
```

CartToPolar

Converts the elements of a complex vector to polar coordinate form.

```
IppStatus ippsCartToPolar_32fc(const Ipp32fc* pSrc, Ipp32f* pDstMagn,
     Ipp32f* pDstPhase, int len);
```

```
IppStatus ippsCartToPolar_64fc(const Ipp64fc* pSrc, Ipp64f* pDstMagn,
     Ipp64f* pDstPhase, int len);
IppStatus ippsCartToPolar_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
     Ipp32f* pDstMagn, Ipp32f* pDstPhase, int len);
IppStatus ippsCartToPolar_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
     Ipp64f* pDstMagn, Ipp64f* pDstPhase, int len);
```

Arguments

pSrc	Pointer to the source vector.
pSrcRe	Pointer to the source vector which stores the real components of Cartesian X,Y pairs.
pSrcIm	Pointer to the source vector which stores the imaginary components of Cartesian X,Y pairs.
pDstMagn	Pointer to the vector which stores the magnitude (radius) component of the elements of the vector *pSrc*.
pDstPhase	Pointer to the vector which stores the phase (angle) component of the elements of the vector *pSrc* in radians. Phase values are in the range (-π,π].
len	Number of elements in the vector.

Discussion

The function ippsCartToPolar converts the elements of a complex input vector *pSrc* or the complex input vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, to polar coordinate form, and stores the magnitude (radius) component of each element in the vector *pDstMagn* and the phase (angle) component of each element in the vector *pDstPhase*.

Example 5-15 verifies that points are lying in the unit radius circle.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-15 Using the ippsCartToPolar Function

```
IppStatus cart2polar( void ) {
     Ipp64f cart[6], magn[4], phase[4];
     int n;
     for (n=0; n<6; ++n) cart[n] = sin(IPP_2PI * n / 8);
     IppStatus st = ippsCartToPolar_64f( cart, cart+2, magn, phase, 4 );
     printf_64f( "magn =", magn, 4, st );
     return st;
}

Output:
    magn =  1.000000 1.000000 1.000000 1.000000
```

PolarToCart

Converts the polar form magnitude/phase pairs stored in input vectors to Cartesian coordinate form.

```
IppStatus ippsPolarToCart_32fc(const Ipp32f* pSrcMagn,
     const Ipp32f* pSrcPhase, Ipp32fc* pDst, int len);
IppStatus ippsPolarToCart_64fc(const Ipp64f* pSrcMagn,
     const Ipp64f* pSrcPhase, Ipp64fc* pDst, int len);

IppStatus ippsPolarToCart_32f(const Ipp32f* pSrcMagn, const
     Ipp32f* pSrcPhase, Ipp32f* pDstRe, Ipp32f* pDstIm, int len);
```

```
IppStatus ippsPolarToCart_64f(const Ipp64f* pSrcMagn, const
     Ipp64f* pSrcPhase, Ipp64f* pDstRe, Ipp64f* pDstIm, int len);
```

Arguments

pSrcMagn	Pointer to the source vector which stores the magnitude (radius) components of the elements in polar coordinate form.
pSrcPhase	Pointer to the vector which stores the phase (angle) components of the elements in polar coordinate form in radians. Phase values are in the range (-π,π].
pDst	Pointer to the resulting vector which stores the complex pairs in Cartesian coordinates (X + iY).
pDstRe	Pointer to the resulting vector which stores the real components of Cartesian X,Y pairs.
pDstIm	Pointer to the resulting vector which stores the imaginary components of Cartesian X,Y pairs.
len	Number of elements in the vectors.

Discussion

The function `ippsPolarToCart` converts the polar form magnitude/phase pairs stored in the input vectors *pSrcMagn* and *pSrcPhase* into a complex vector and stores the results in the vector *pDst*, or stores the real components of the result in the vector *pDstRe* and the imaginary components in the vector *pDstIm*.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

MaxOrder

Computes the maximum order of a vector.

```
IppStatus ippsMaxOrder_16s(const Ipp16s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32s(const Ipp32s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32f(const Ipp32f* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_64f(const Ipp64f* pSrc, int len, int* pOrder);
```

Arguments

pSrc	Pointer to the source vector.
len	Number of elements in the vector.
pOrder	Pointer to the result value.

Discussion

The function `ippsMaxOrder` finds the maximum binary number in elements of the exponent vector *pSrc*, and stores the result in *pOrder*.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pOrder* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsNanArg`	Indicates a warning when `NaN` is encountered in the input data vector.

Preemphasize

Computes preemphasis of a single precision real signal in-place.

```
IppStatus ippsPreemphasize_16s(Ipp16s* pSrcDst, int len, Ipp32f val);
IppStatus ippsPreemphasize_32f(Ipp32f* pSrcDst, int len, Ipp32f val);
```

Arguments

pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
val	Multiplier factor used in the difference signal preemphasis equation.

Discussion

The in-place function `ippsPreemphasize` computes preemphasis of a real signal *pSrcDst*. The computation is performed according to the difference signal preemphasis equation:

$y(n) = x(n) - val \cdot x(n - 1)$,

where $y(n)$ is the preemphasized output, $x(n)$ is the input, and *val* is the multiplier factor.

Note that usually *val* = 0.95 for speech signals.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Flip

Reverses the order of elements in a vector.

```
IppStatus ippsFlip_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsFlip_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsFlip_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsFlip_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsFlip_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsFlip_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsFlip_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsFlip_64f_I(Ipp64f* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsFlip stores the elements of a source vector *pSrc* to a destination vector *pDst* in reverse order according to the following formula:

pDst[*n*] = *pSrc*[*len*-*n*-1], *n*=0..*len*-1

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

FindNearestOne

Finds an element of the table which is closest to the specified value.

```
IppStatus ippsFindNearestOne_16u(Ipp16u inpVal, Ipp16u* pOutVal,
     int* pOutIndex, const Ipp16u *pTable, int tblLen);
```

Arguments

inpVal	Reference value.
pOutVal	Pointer to the output value.
pOutIndx	Pointer to the output index.
pTable	Pointer to the table for searching.
tblLen	Number of elements in the table.

Discussion

The ippsFindNearestOne function searches through the table *pTable* for an element which is closest to the specified reference value *inpVal*. The resulting element and its index are stored in *pOutVal* and *pOutIndex*, respectively.
The table elements should satisfy the condition $pTable[i] \leq pTable[i+1]$.
The function uses the following distance criterion for determining the closest element: min(|*inpVal*-*pTable*[*i*]|).

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *tblLen* is less than or equal to 0.

FindNearest

Finds table elements that are closest to the elements of the specified vector.

```
IppStatus ippsFindNearest_16u(const Ipp16u* pVals, Ipp16u* pOutVals,
     int* pOutIndexes, int len, const Ipp16u *pTable, int tblLen);
```

Arguments

pVals	Pointer to the vector containing reference values.
pOutVals	Pointer to the output vector.
pOutIndexes	Pointer to the array that stores output indexes.
len	Number of elements in the input vector.
pTable	Pointer to the table for searching.
tblLen	Number of elements in the table.

Discussion

The ippsFindNearest function searches through the table *pTable* for elements which are closest to the reference elements of the input vector *pVals*. The resulting elements and their indexes are stored in *pOutVals* and *pOutIndexes*, respectively.
The table elements should satisfy the condition *pTable*[*i*] ≤ *pTable*[*i+1*].
The function uses the following distance criterion for determining the table element closest to *pVal*[*k*] :
min(|*pVals[k]-pTable*[i]|).

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *tblLen* or *len* is less than or equal to 0.

Companding Functions

The functions described in this section perform an operation of data compression by using a logarithmic encoder-decoder, referred to as companding. Companding allows you to maintain a constant percentage error by logarithmically spacing the quantization levels [Rab78].

The IPP companding functions perform the following conversion operations of signal samples:

- From 8-bit μ-law encoded format to PCM-linear or vice-versa.
- From 8-bit A-law encoded format to PCM-linear or vice-versa.
- From 8-bit μ-law encoded format to A-law encoded or vice-versa.

Samples encoded in μ-law or A-law format are non-uniformly quantized. The quantization functions used by these formats are designed to reduce the dependency of signal-to-noise ratio on the magnitude of the encoded signal. This is achieved by quantization (companding) at a finer resolution near zero, and at a coarse resolution at larger positive or negative levels. The output values are normalized to be in the range [-1; +1].

These functions perform the μ-law and A-law companding in compliance with the CCITT G.711 specification, [CCITT]. For the conversion rules and more details, refer to [CCITT].

Example 5-16 shows how to use companding functions.

MuLawToLin

Decodes samples from 8-bit μ-law encoded format to linear samples.

```
IppStatus ippsMuLawToLin_8u16s(const Ipp8u* pSrc, Ipp16s* pDst, int len);
IppStatus ippsMuLawToLin_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector, which stores 8-bit μ-law encoded signal samples to be decoded.
pDst	Pointer to the destination vector, which stores the linear sample results.
len	Number of samples in the vector.

Discussion

The function ippsMuLawToLin decodes the 8-bit μ-law encoded samples in the vector *pSrc* to PCM-linear samples and stores them in the vector *pDst*.

The formula for μ-law companding is as follows:

$$\left|C_{\mu}(x)\right| = \frac{\ln(1 + 255 \cdot |x|)}{\ln(256)} \cdot 128, \quad -1 \le x \le 1$$

where x is the linear signal sample and $C_{\mu}(x)$ is the μ-law encoded sample.

The formula is shown in terms of absolute values of both the original and compressed signals since positive and negative values are compressed in an identical manner. The sign of the input is preserved in the output.

Application Notes

The formula shown above should not be implemented directly, since such an implementation would be slow. Encoding or decoding of μ-law format is usually performed using look-up Tables 2a/G.711 and 2b/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

LinToMuLaw

Encodes the linear samples using 8-bit μ-law format and stores them in a vector.

```
IppStatus ippsLinToMuLaw_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLinToMuLaw_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len);
```

Arguments

pSrc	Pointer to the vector that holds the signal samples (normalized to be less than 1.0) to be encoded.
pDst	Pointer to the vector that holds the output of the function ippsLinToMuLaw.
len	Number of samples in the vector.

Discussion

The function ippsLinToMuLaw encodes the PCM-linear samples in the vector *pSrc* using 8-bit μ-law format and stores them in the vector *pDst*.

Example 5-16 shows how to use the function ippsLinToMuLaw_32f8u.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

ALawToLin

Decodes the 8-bit A-law encoded samples to linear samples.

```
IppStatus ippsALawToLin_8u16s(const Ipp8u* pSrc, Ipp16s* pDst, int len);
IppStatus ippsALawToLin_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the vector that holds the signal samples to be converted.
pDst	Pointer to the vector that holds the output of the function ippsALawToLin.
len	Number of samples in the vector.

Discussion

The function ippsALawToLin decodes the 8-bit A-law encoded samples in the vector *pSrc* to PCM-linear samples and stores them in the vector *pDst*.

The formula for A-law companding is as follows:

$$|C_A(x)| = \begin{cases} \frac{87.56|x|}{1+\ln 87.56} \cdot 128, & 0 \le |x| \le \frac{1}{87.56} \\ \frac{1+\ln(87.56|x|)}{1+\ln 87.56} \cdot 128, & \frac{1}{87.56} < |x| \le 1 \end{cases},$$

where x is the linear signal sample and $C_A(x)$ is the A-law encoded sample.

The formula is shown in terms of absolute values of both the original and compressed signals since positive and negative values are compressed in an identical manner. The sign of the input is preserved in the output.

Application Notes

The formula shown above should not be implemented directly, since such an implementation would be slow. Encoding or decoding of A-law format is usually performed using look-up Tables 1a/G.711 and 1b/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Example 5-16 shows how to use the function `ippsALawToLin_8u32f`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pDst`* or *`pSrc`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.

LinToALaw

Encodes the linear samples using 8-bit A-law format and stores them in an array.

```
IppStatus ippsLinToALaw_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLinToALaw_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len);
```

Arguments

`pSrc`	Pointer to the vector that holds the signal samples to be encoded.
`pDst`	Pointer to the vector that holds the output of the function `ippsLinToALaw`.
`len`	Number of samples in the vector.

Discussion

The function `ippsLinToALaw` encodes the PCM-linear samples in the vector *`pSrc`* using 8-bit A-law format and stores them in the vector *`pDst`*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

MuLawToALaw

Converts samples from 8-bit μ-law encoded format to 8-bit A-law encoded format.

```
IppStatus ippsMuLawToALaw_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector, which stores 8-bit μ-law encoded signal samples.
pDst	Pointer to the destination vector, which stores the 8-bit A-law encoded samples.
len	Number of samples in the vector.

Discussion

The function ippsMuLawToALaw converts signal samples from 8-bit μ-law encoded format in the vector *pSrc* to 8-bit A-law encoded format and stores them in the vector *pDst*.

Application Notes

The conversion of μ-law format to A-law format is usually performed using look-up Table 3/G.711shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Example 5-16 shows how to use the function ippsMuLawToALaw_8u.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

ALawToMuLaw

Converts samples from 8-bit A-law encoded format to 8-bit μ-law encoded format.

```
IppStatus ippsALawToMuLaw_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector, which stores 8-bit A-law encoded signal samples.
pDst	Pointer to the destination vector, which stores the 8-bit μ-law encoded samples.
len	Number of samples in the vector.

Discussion

The function ippsMuLawToALaw converts signal samples from 8-bit A-law encoded format in the vector *pSrc* to 8-bit μ-law format and stores them in the vector *pDst*.

Application Notes

The conversion of A-law format to μ-law format is usually performed using look-up Table 4/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-16 Using Companding Functions

```
void compand( void ) {
     Ipp32f x[4] = { 0.1f, 0.2f, 0.3f, 0.4f };
     Ipp8u m[4], a[4];
     ippsLinToMuLaw_32f8u( x, m, 4 );
     ippsMuLawToALaw_8u( m, a, 4 );
     ippsALawToLin_8u32f( a, x, 4 );
     // now x must be close to original
     printf_32f("x =", x, 4, ippStsNoErr);
}
Output:
    x =  0.099609 0.207031 0.304688 0.398438
```

Windowing Functions

This chapter describes several of the windowing functions commonly used in signal processing. A window is a mathematical function by which a signal is multiplied to improve the characteristics of some subsequent analysis. Windows are commonly used in FFT-based spectral analysis.

Understanding Window Functions

The IPP provides the following functions to generate window samples:

- Bartlett windowing function
- Blackman family of windowing functions
- Hamming windowing function
- Hann windowing function
- Kaiser windowing function

These functions generate the window samples and multiply them into an existing signal. To obtain the window samples themselves, initialize the vector argument to the unity vector before calling the window function.

If you want to multiply different frames of a signal by the same window multiple times, it is better to first calculate the window by calling one of the windowing functions (`ippsWinHann`, for example) on a vector with all elements set to 1.0. Then use one of the vector multiplication functions (`ippsMul`, for example) to multiply the window into the signal each time a new set of input samples is available. This avoids repeatedly calculating the window samples. This is illustrated in Example 5-17.

Example 5-17 Window and FFT Many Frames of a Signal

```
void multiFrameWin( void ) {
     Ipp32f win[LEN], x[LEN], X[LEN];
     IppsFFTSpec_R_32f* ctx;
     ippsSet_32f( 1, win, LEN );
     ippsWinHann_32f_I( win, LEN );
     /// ... initialize FFT context
     while(1 ){
        /// ... get x signal
        ///
        ippsMul_32f_I( win, x, LEN );
        ippsFFTFwd_RToPack_32f( x, X, ctx, 0 );
     }
}
```

Related Topics

For more information on windowing, see: [Jac89], section 7.3, *Windows in Spectrum Analysis*; [Jac89], section 9.1, *Window-Function Technique*; and [Mit93], section 16-2, *Fourier Analysis of Finite-Time Signals*. For more information on these references, see also the Bibliography at the end of this manual.

WinBartlett

Multiplies a vector by a Bartlett windowing function.

```
IppStatus ippsWinBartlett_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len);
IppStatus ippsWinBartlett_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len);
IppStatus ippsWinBartlett_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len);
IppStatus ippsWinBartlett_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len);
IppStatus ippsWinBartlett_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBartlett_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBartlett_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBartlett_32fc_I(Ipp32fc* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsWinBartlett multiplies the vector *pSrc* by the Bartlett (triangle) window, and stores the result in *pDst*.

The in-place flavors of ippsWinBartlett multiply the *pSrcDst* by the Bartlett (triangle) window and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window.

The Bartlett window is defined as follows:

$$w_{bartlett}(n) = \begin{cases} \frac{2n}{len-1}, & 0 \le n \le \frac{len-1}{2} \\ 2 - \frac{2n}{len-1}, & \frac{len-1}{2} < n \le len-1 \end{cases}$$

Example 5-18 shows how to use the function `ippsWinBartlett_32f_I`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrc`*, *`pDst`*, *`or pSrcDst`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than 3.

Example 5-18 Using the ippsWinBartlett Function

```
void bartlett(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinBartlett_32f_I(x, 8);
      printf_32f("bartlett (half) =", x, 4, ippStsNoErr);
}
Output:
    bartlett (half) =  0.000000 0.285714 0.571429 0.857143
Matlab* Analog:
    >> b = bartlett(8); b(1:4)'
```

WinBlackman

Multiplies a vector by a Blackman windowing function.

```
IppStatus ippsWinBlackmanQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst,
      int len, int alphaQ15);
IppStatus ippsWinBlackmanQ15_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
      int len, int alphaQ15);
IppStatus ippsWinBlackman_16s(const Ipp16s* pSrc, Ipp16s* pDst,
      int len, float alpha);
IppStatus ippsWinBlackman_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
      int len, float alpha);
IppStatus ippsWinBlackman_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int len, float alpha);
IppStatus ippsWinBlackman_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      int len, float alpha);
IppStatus ippsWinBlackmanQ15_16s_I(Ipp16s* pSrcDst, int len,
      int alphaQ15);
IppStatus ippsWinBlackmanQ15_16sc_I(Ipp16sc* pSrcDst, int len,
      int alphaQ15);
IppStatus ippsWinBlackman_16s_I(Ipp16s* pSrcDst, int len,
      float alpha);
IppStatus ippsWinBlackman_16sc_I(Ipp16sc* pSrcDst, int len,
      float alpha);
IppStatus ippsWinBlackman_32f_I(Ipp32f* pSrcDst, int len,
      float alpha);
IppStatus ippsWinBlackman_32fc_I(Ipp32fc* pSrcDst, int len,
      float alpha);
IppStatus ippsWinBlackmanStd_16s(const Ipp16s* pSrc, Ipp16s* pDst,
      int len);
IppStatus ippsWinBlackmanStd_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
      int len);
IppStatus ippsWinBlackmanStd_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int len);
```

```
IppStatus ippsWinBlackmanStd_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len);
IppStatus ippsWinBlackmanStd_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len);
IppStatus ippsWinBlackmanOpt_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len);
IppStatus ippsWinBlackmanOpt_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len);
IppStatus ippsWinBlackmanOpt_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len);
IppStatus ippsWinBlackmanOpt_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_32fc_I(Ipp32fc* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
alpha	Adjustable parameter associated with the Blackman windowing equation.
alphaQ15	Scaled version of *alpha*. The *scaleFactor* value is 15.
len	Number of elements in the vector.

Discussion

The ippsWinBlackman family of functions multiply the vector *pSrc* by the Blackman window, and stores the result in *pDst*.

The in-place ippsWinBlackman family of functions multiply the vector *pSrcDst* by the Blackman window, and stores the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The functions for the Blackman family of windows are defined below.

ippsWinBlackman. The function ippsWinBlackman allows the application to specify *alpha*. The Blackman window is defined as follows:

$$w_{blackman}(n) = \frac{alpha+1}{2} - 0.5\cos\left(\frac{2\pi n}{len-1}\right) - \frac{alpha}{2}\cos\left(\frac{4\pi n}{len-1}\right)$$

ippsWinBlackmanQ15. The function ippsWinBlackmanQ15 multiplies a vector by a Blackman window with *alphaQ15* scaled according to the factor 15.

ippsWinBlackmanStd. The standard Blackman window is provided by the function ippsWinBlackmanStd, which simply multiplies a vector by a Blackman window with the standard value of *alpha* shown below:

alpha = -0.16

ippsWinBlackmanOpt. The function ippsWinBlackmanOpt provides a modified window that has a 30 dB/octave roll-off by multiplying a vector by a Blackman window with the optimal value of *alpha* shown below:

$$alpha = -\frac{0.5}{1+\cos\frac{2\pi}{len-1}}$$

The minimum *len* is equal to 4. For large *len*, the optimal *alpha* converges asymptotically to the asymptotic *alpha*; the application can use the asymptotic value of *alpha* shown below:

alpha = -0.25

Example 5-19 shows how to use the function ippsWinBlackmanStd_32f_I

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.

`ippStsSizeErr`	Indicates an error when *`len`* is less than 4 for the function `ippsWinBlackmanOpt` and less than 3 for all other functions of the family.

Example 5-19 Using the ippsWinBlackmanStd Function

```
void blackman(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinBlackmanStd_32f_I(x, 8);
      printf_32f("blackman (half) =", x, 4, ippStsNoErr);
}
Output:
    blackman(half) =  0.000000 0.090453 0.459183 0.920364
Matlab* Analog:
    >> b = blackman(8)'; b(1:4)
```

WinHamming

Multiplies a vector by a Hamming windowing function.

```
IppStatus ippsWinHamming_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinHamming_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinHamming_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinHamming_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinHamming_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHamming_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHamming_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHamming_32fc_I(Ipp32fc* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsWinHamming multiplies the vector *pSrc* by the Hamming window and stores the result in *pDst*.

The in-place flavors of ippsWinHamming multiply the vector *pSrcDst* by the Hamming window and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The Hamming window is defined as follows:

$$w_{hamming}(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{len-1}\right)$$

Example 5-20 shows how to use the function ippsWinHamming_32f_I

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than 3.

Example 5-20 Using the ippsWinHamming Function

```
void hamming(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinHamming_32f_I(x, 8);
      printf_32f("hamming(half) =", x, 4, ippStsNoErr);
}
Output:
    hamming(half) =  0.080000 0.253195 0.642360 0.954446
Matlab* Analog:
    >> b = hamming(8); b(1:4)'
```

WinHann

Multiplies a vector by a Hann windowing function.

```
IppStatus ippsWinHann_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinHann_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinHann_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinHann_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinHann_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHann_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHann_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHann_32fc_I(Ipp32fc* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Discussion

The function ippsWinHann multiplies the vector *pSrc* by the Hann window, and stores the result in *pDst*.

The in-place flavors of ippsWinHann multiply the vector *pSrcDst* by the Hann window and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The Hann window is defined as follows:

$$w_{hann}(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{len-1}\right)$$

Example 5-21 shows how to use the function ippsWinHann_32f_I

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than 3.

Example 5-21 Using the ippsWinHann Function

```
void hann(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinHann_32f_I(x, 8);
      printf_32f("hann(half) =", x, 4, ippStsNoErr);
}
Output:
    hann(half) =  0.000000 0.188255 0.611260 0.950484
Matlab* Analog:
    >> N = 8; n = 0:N-1; 0.5*(1-cos(2*pi*n/(N-1)))
```

WinKaiser

Multiplies a vector by a Kaiser windowing function.

```
IppStatus ippsWinKaiser_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
      float alpha);
IppStatus ippsWinKaiser_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
      float alpha);
IppStatus ippsWinKaiser_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
      float alpha);
IppStatus ippsWinKaiser_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
      int len, float alpha);
IppStatus ippsWinKaiser_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      int len, float alpha);
IppStatus ippsWinKaiserQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst,
      int len, int alphaQ15);
IppStatus ippsWinKaiserQ15_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
      int len, int alphaQ15);
```

```
IppStatus ippsWinKaiser_16s_I(Ipp16s* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_32f_I(Ipp32f* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_64f_I(Ipp64f* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_16sc_I(Ipp16sc* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_32fc_I(Ipp32fc* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiserQ15_16s_I(Ipp16s* pSrcDst, int len, int alphaQ15);
IppStatus ippsWinKaiserQ15_16sc_I(Ipp16sc* pSrcDst, int len, int alphaQ15);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
alpha	Adjustable parameter associated with the Kaiser windowing equation.
alphaQ15	Scaled version of *alpha*. The *scaleFactor* value is 15.
len	Number of elements in the vector.

Discussion

The function ippsWinKaiser multiplies the vector *pSrc* by the Kaiser window, and stores the result in *pDst*.

The in-place flavors of ippsWinKaiser multiply the vector *pSrcDst* by the Kaiser window and store the result in *pSrcDst*.

ippsWinKaiser. The function ippsWinKaiser allows the application to specify *alpha*. The function multiplies both real and imaginary parts of the complex vector by the same window. The Kaiser family of windows are defined as follows:

$$w_{kaiser}(n) = \frac{I_0\left(alpha\sqrt{\left(\frac{len-1}{2}\right)^2-\left(n-\left(\frac{len-1}{2}\right)\right)^2}\right)}{I_0\left(alpha\left(\frac{len-1}{2}\right)\right)}$$

Here $I_0()$ is the modified zero-order Bessel function of the first kind.

ippsWinKaiserQ15. The function ippsWinKaiserQ15 multiplies a vector by a Kaiser window with *alphaQ15* scaled according to the factor 15.

Example 5-22 shows how to use the function ippsWinKaiser_32f_I

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst*, *pSrc*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than 1.
ippStsHugeWinErr	Indicates an error when the Kaiser window is too big.

Example 5-22 Using the ippsWinKaiser Function

```
void kaiser(void) {
     Ipp32f x[8];
     IppStatus st;
     ippsSet_32f(1, x, 8);
     st = ippsWinKaiser_32f_I( x, 8, 1.0f );
     printf_32f("kaiser(half) =", x, 4, ippStsNoErr);
}
Output:
   kaiser(half) =  0.135534 0.429046 0.755146 0.970290
Matlab* Analog:
    >> kaiser(8,7/2)'
```

Statistical Functions

This section describes the IPP functions that compute the vector measure values: maximum, minimum, mean, and standard deviation.

Sum

Computes the sum of the elements of a vector.

```
IppStatus ippsSum_32f(const Ipp32f* pSrc, int len, Ipp32f* pSum,
      IppHintAlgorithm hint);
IppStatus ippsSum_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pSum,
      IppHintAlgorithm hint);
IppStatus ippsSum_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSum_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pSum,
      int scaleFactor);
IppStatus ippsSum_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pSum,
      int scaleFactor);
IppStatus ippsSum_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pSum,
      int scaleFactor);
IppStatus ippsSum_16sc32sc_Sfs(const Ipp16sc* pSrc, int len, Ipp32sc* pSum,
      int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pSum	Pointer to the output result.
len	Number of elements in the vector.
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsSum computes the sum of the elements of the vector *pSrc* and stores the result in *pSum*.

The sum of the elements of *pSrc* is defined by the formula:

$$sum = \sum_{n=0}^{len-1} pSrc[n]$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

When computing the sum of integer numbers, the output result can exceed the data range and become saturated. To get a precise result, use the scale factor. The scaling is performed in accordance with the *scaleFactor* value.

Example 5-23 shows how to use the function ippsSum.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSum* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-23 Using the ippsSum Function

```
void sum(void) {
      Ipp16s x[4] = {-32768, 32767, 32767, 32767}, sm;
      ippsSum_16s_Sfs(x, 4, &sm, 1);
      printf_16s("sum =", &sm, 1, ippStsNoErr);
}
Output:
    sum = 32766
Matlab* Analog:
    >> x = [-32768, 32767, 32767, 32767]; sum(x)/2
```

Max

Returns the maximum value of a vector.

```
IppStatus ippsMax_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax);
IppStatus ippsMax_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax);
IppStatus ippsMax_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax);
```

Arguments

pSrc	Pointer to the source vector.
pMax	Pointer to the output result.
len	Number of elements in the vector

Discussion

The function ippsMax returns the maximum value of the input vector *pSrc*, and stores the result in *pMax*.

Example 5-24 shows how to use the function ippsMax_32f.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMax* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

MaxIndx

Returns the maximum value of a vector and the index of the maximum element.

```
IppStatus ippsMaxIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax,
     int* pIndx);
```

```
IppStatus ippsMaxIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax,
     int* pIndx);
IppStatus ippsMaxIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax,
     int* pIndx);
```

Arguments

pSrc	Pointer to the source vector.
pMax	Pointer to the output result.
len	Number of elements in the vector.
pIndx	Pointer to the index value of the maximum element.

Discussion

The function ippsMaxIndx returns the maximum value of the input vector *pSrc*, and stores the result in *pMax*. If *pIndx* is not a NULL pointer, the function returns the index of the maximum element and stores it in *pIndx*. If there are several equal maximum elements, the first index from the beginning is returned.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMax* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Min

Returns the minimum value of a vector.

```
IppStatus ippsMin_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin);
IppStatus ippsMin_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin);
IppStatus ippsMin_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin);
```

Arguments

pSrc	Pointer to the source vector.
pMin	Pointer to the output result.
len	Number of elements in the vector.

Discussion

The function ippsMin returns the minimum value of the input vector *pSrc*, and stores the result in *pMin*.

Example 5-24 shows how to use the function ippsMin_32f.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMin* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

MinIndx

Returns the minimum value of a vector and the index of the minimum element.

```
IppStatus ippsMinIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin,
     int* pIndx);
IppStatus ippsMinIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin,
     int* pIndx);
IppStatus ippsMinIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin,
     int* pIndx);
```

Arguments

pSrc	Pointer to the source vector.
pMin	Pointer to the output result.

len	Number of elements in the vector.
pIndx	Pointer to the index value of the minimum element.

Discussion

The function ippsMinIndx returns the minimum value of the input vector *pSrc* and stores the result in *pMin*. If *pIndx* is not a NULL pointer, the function returns the index of the minimum element and stores it in *pIndx*. If there are several equal minimum elements, the first index from the beginning is returned.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMin* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-24 Using the ippsMin and ippsMax Functions

```
void minmax(void) {
      Ipp32f *x = ippsMalloc_32f(1000), minmax[2];
      int i;
      for (i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
      ippsMin_32f(x, 1000, &minmax[0]);
      ippsMax_32f(x, 1000, &minmax[1]);
      printf_32f("min max =", minmax, 2, ippStsNoErr);
      ippsFree(x);
}
Output:
    min max = 0.000855 0.999695
Matlab* Analog:
    >> x = rand(1,1000); min(x), max(x)
```

Mean

Computes the mean value of a vector.

```
IppStatus ippsMean_32f(const Ipp32f* pSrc, int len, Ipp32f* pMean,
     IppHintAlgorithm hint);
IppStatus ippsMean_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pMean,
     IppHintAlgorithm hint);
IppStatus ippsMean_64f(const Ipp64f* pSrc, int len, Ipp64f* pMean);
IppStatus ippsMean_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pMean,
     int scaleFactor);
IppStatus ippsMean_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pMean,
     int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pMean	Pointer to the output result.
len	Number of elements in the vector
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsMean computes the mean (average) of the vector *pSrc*, and stores the result in *pMean*. The mean of *pSrc* is defined by the formula:

$$mean = \frac{1}{len} \sum_{n=0}^{len-1} pSrc[n]$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

Example 5-25 shows how to use the function ippsMean_32f

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMean* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-25 Using the ippsMean Function

```
void mean(void) {
      Ipp32f *x = ippsMalloc_32f(1000), mean;
      int i;
      for(i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
      ippsMean_32f(x, 1000, &mean, ippAlgHintFast);
      printf_32f("mean =", &mean, 1, ippStsNoErr);
      ippsFree(x);
}
Output:
    mean = 0.492591
Matlab* Analog:
    >> x = rand(1,1000); mean(x)
```

StdDev

Computes the standard deviation value of a vector.

```
IppStatus ippsStdDev_32f(const Ipp32f* pSrc, int len, Ipp32f* pStdDev,
     IppHintAlgorithm hint);
IppStatus ippsStdDev_64f(const Ipp64f* pSrc, int len, Ipp64f* pStdDev);
IppStatus ippsStdDev_16s32s_Sfs(const Ipp16s* pSrc, int len,
     Ipp32s* pStdDev, int scaleFactor);
```

```
IppStatus ippsStdDev_16s_Sfs(const Ipp16s* pSrc, int len,
      Ipp16s* pStdDev, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pStdDev	Pointer to the output result.
len	Number of elements in the vector
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsStdDev computes the standard deviation of the input vector *pSrc*, and stores the result in *pStdDev*. The vector length can not be less than 2. The standard deviation of *pSrc* is defined by the unbiased estimate formula:

$$stdev = \sqrt{\frac{\sum_{n=0}^{len-1} (pSrc[n] - mean(pSrc))^2}{len-1}}$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

Example 5-26 shows how to use the function ippsStdDev_32f.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pStdDev* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 1.

Example 5-26 Using the ippsStdDev Function

```
void stdev(void) {
      Ipp32f *x = ippsMalloc_32f(1000), stdev;
      int i;
      for (i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
      ippsStdDev_32f(x, 1000, &stdev, ippAlgHintFast);
      printf_32f("stdev =", &stdev, 1, ippStsNoErr);
      ippsFree(x);
}
Output:
    stdev = 0.286813
Matlab* Analog:
    >> x = rand(1,1000); std(x)
```

Norm

Computes the C, L1, or L2 norm of a vector.

```
IppStatus ippsNorm_Inf_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_Inf_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_16s32s_Sfs(const Ipp16s* pSrc, int len,
     Ipp32s* pNorm, int scaleFactor);
IppStatus ippsNorm_L1_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L1_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L1_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L1_16s32s_Sfs(const Ipp16s* pSrc, int len,
     Ipp32s* pNorm, int scaleFactor);
IppStatus ippsNorm_L2_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L2_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
```

```
IppStatus ippsNorm_L2_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L2_16s32s_Sfs(const Ipp16s* pSrc, int len,
      Ipp32s* pNorm, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector *pSrc*.
pNorm	Pointer to the output result.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsNorm computes the C, L1, or L2 norm of the source vector *pSrc* and stores the result in *pNorm*.

ippsNorm_Inf. The function ippsNorm_Inf computes the C norm defined by the formula:

$$Norm_C = \max_{n=0}^{len-1} |pSrc[n]|$$

ippsNorm_L1. The function ippsNorm_L1 computes the L1 norm defined by the formula:

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc[n]|$$

ippsNorm_L2. The function ippsNorm_L2 computes the L2 norm defined by the formula:

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc[n]|^2}$$

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrc* or *pNorm* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

NormDiff

Computes the C, L1, or L2 norm of two vectors' difference.

```
IppStatus ippsNormDiff_Inf_32f(const Ipp32f* pSrc1,
     const Ipp32f* pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_64f(const Ipp64f* pSrc1,
     const Ipp64f* pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_Inf_16s32f(const Ipp16s* pSrc1,
     const Ipp16s* pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_16s32s_Sfs(const Ipp16s* pSrc1,
     const Ipp16s* pSrc2, int len, Ipp32s* pNorm, int scaleFactor);
IppStatus ippsNormDiff_L1_32f(const Ipp32f* pSrc1,
     const Ipp32f* pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_L1_64f(const Ipp64f* pSrc1,
     const Ipp64f* pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L1_16s32f(const Ipp16s* pSrc1,
     const Ipp16s* pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_L1_16s32s_Sfs(const Ipp16s* pSrc1,
     const Ipp16s* pSrc2, int len, Ipp32s* pNorm, int scaleFactor);
IppStatus ippsNormDiff_L2_32f(const Ipp32f* pSrc1,
     const Ipp32f* pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_L2_64f(const Ipp64f* pSrc1,
     const Ipp64f* pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L2_16s32f(const Ipp16s* pSrc1,
     const Ipp16s* pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_L2_16s32s_Sfs(const Ipp16s* pSrc1,
     const Ipp16s* pSrc2, int len, Ipp32s* pNorm, int scaleFactor);
```

Arguments

pSrc1, pSrc2	Pointers to the two source vectors; *pSrc2* can be NULL.
pNorm	Pointer to the output result.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsNorm computes the C, L1, or L2 norm of the source vectors' difference, and stores the result in *pNorm*.

ippsNormDiff_Inf. The function ippsNormDiff_Inf computes the C norm defined by the formula:

$$Norm_{Inf} = \max_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

ippsNormDiff_L1. The function ippsNormDiff_L1 computes the L1 norm defined by the formula:

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

ippsNormDiff_L2. The function ippsNormDiff_L2 computes the L2 norm defined by the formula:

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|^2}$$

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value.

Example 5-27 shows how to use the function ippsNormDiff.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, or *pNorm* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-27 Using the ippsNorm Function

```
int norm( void ) {
      Ipp16s x[LEN];
      Ipp32f Norm[3];
      IppStatus st;
      int i;
      for( i=0; i<LEN; ++i ) x[i] = (Ipp16s)rand();
      ippsNormDiff_Inf_16s32f( x, 0, LEN, Norm );
      ippsNormDiff_L1_16s32f( x, 0, LEN, Norm+1 );
      st = ippsNormDiff_L2_16s32f( x, 0, LEN, Norm+2 );
      printf_32f("Norm (oo,L1,L2) =", Norm, 3, st );
      return Norm[2] <= Norm[1] && Norm[1] <= LEN*Norm[0];
}
Output:
    Norm (oo,L1,L2) =  31993.000000 1526460.000000 180270.781250
Matlab* analog:
    >> x = 32767*rand(1,100);norm(x,inf),norm(x,1),norm(x,2)
```

DotProd

Computes the dot product of two vectors.

```
IppStatus ippsDotProd_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
      int len, Ipp32f* pDp);
IppStatus ippsDotProd_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
      int len, Ipp32fc* pDp);
IppStatus ippsDotProd_32f32fc(const Ipp32f* pSrc1, const Ipp32fc* pSrc2,
      int len, Ipp32fc* pDp);
IppStatus ippsDotProd_32f64f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
      int len, Ipp64f* pDp);
```

```
IppStatus ippsDotProd_32fc64fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
     int len, Ipp64fc* pDp);

IppStatus ippsDotProd_32f32fc64fc(const Ipp32f* pSrc1,
     const Ipp32fc* pSrc2, int len, Ipp64fc* pDp);

IppStatus ippsDotProd_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
     int len, Ipp64f* pDp);

IppStatus ippsDotProd_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
     int len, Ipp64fc* pDp);

IppStatus ippsDotProd_64f64fc(const Ipp64f* pSrc1, const Ipp64fc*
     pSrc2, int len, Ipp64fc* pDp);

IppStatus ippsDotProd_16s64s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
     int len, Ipp64s* pDp);

IppStatus ippsDotProd_16sc64sc(const Ipp16sc* pSrc1, const Ipp16sc*
     pSrc2, int len, Ipp64sc* pDp);

IppStatus ippsDotProd_16s16sc64sc(const Ipp16s* pSrc1, const Ipp16sc*
     pSrc2, int len, Ipp64sc* pDp);

IppStatus ippsDotProd_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
     int len, Ipp32f* pDp);

IppStatus ippsDotProd_16sc32fc(const Ipp16sc* pSrc1, const Ipp16sc*
     pSrc2, int len, Ipp32fc* pDp);

IppStatus ippsDotProd_16s16sc32fc(const Ipp16s* pSrc1, const Ipp16sc*
     pSrc2, int len, Ipp32fc* pDp);

IppStatus ippsDotProd_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
     pSrc2, int len, Ipp16s* pDp, int scaleFactor);

IppStatus ippsDotProd_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc*
     pSrc2, int len, Ipp16sc* pDp, int scaleFactor);

IppStatus ippsDotProd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s*
     pSrc2, int len, Ipp32s* pDp, int scaleFactor);

IppStatus ippsDotProd_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc*
     pSrc2, int len, Ipp32sc* pDp, int scaleFactor);

IppStatus ippsDotProd_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
     pSrc2, int len, Ipp32s* pDp, int scaleFactor);

IppStatus ippsDotProd_16s16sc32sc_Sfs(const Ipp16s* pSrc1, const
     Ipp16sc* pSrc2, int len, Ipp32sc* pDp, int scaleFactor);

IppStatus ippsDotProd_16s32s32s_Sfs(const Ipp16s* pSrc1, const Ipp32s*
     pSrc2, int len, Ipp32s* pDp, int scaleFactor);
```

```
IppStatus ippsDotProd_16s16sc_Sfs(const Ipp16s* pSrc1, const Ipp16sc*
     pSrc2, int len, Ipp16sc* pDp, int scaleFactor);
IppStatus ippsDotProd_16sc32sc_Sfs(const Ipp16sc* pSrc1, const
     Ipp16sc* pSrc2, int len, Ipp32sc* pDp, int scaleFactor);
IppStatus ippsDotProd_32s32sc_Sfs(const Ipp32s* pSrc1, const Ipp32sc*
     pSrc2, int len, Ipp32sc* pDp, int scaleFactor);
```

Arguments

pSrc1	Pointer to the first vector to compute the dot product value.
pSrc2	Pointer to the second vector to compute the dot product value.
pDp	Pointer to the output result.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsDotProd computes the dot product (scalar value) of two vectors, *pSrc1* and *pSrc2*, and stores the result in *pDp*.

The computation is performed as follows:

$$dp = \sum_{n=0}^{len-1} pSrc1[n] \cdot pSrc2[n]$$

To compute the dot product of complex data, use the function ippsConj to conjugate one of the operands. The vectors *pSrc1* and *pSrc2* must be of equal length.

Example 5-28 shows how to use the function ippsDotProd_64f to verify orthogonality of the sine and cosine functions. Two vectors are orthogonal to each other when the dot product of the two vectors is zero.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pDp*, *pSrc1*, or *pSrc*2 pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-28 Using ippsDotProd to Verify Orthogonality of Sin and Cos

```
void dotprod(void) {
      Ipp64f x[10], dp;
      int n;
      for (n = 0; n<10; ++n) x[n] = sin(IPP_2PI * n / 8);
      ippsDotProd_64f(x, x+2, 8, &dp);
      printf_64f("dp =", &dp, 1, ippStsNoErr);
}
Output:
    dp = 0.000000
Matlab* Analog:
    >> n = 0:9; x = sin(2*pi*n/8); a = x(1:8); b = x(3:10); a*b'
```

MaxEvery, MinEvery

Computes maximum or minimum value for each pair of elements of two vectors.

```
IppStatus ippsMaxEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMaxEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len);
IppStatus ippsMaxEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMinEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMinEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len);
IppStatus ippsMinEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the first input vector.

pSrcDst	Pointer to the second input vector which stores the result.
len	Number of elements in the vector.

Discussion

The function ippsMaxEvery computes the maximum between each pair of corresponding elements of two input vectors and stores the result in *pSrcDst*. The function ippsMinEvery computes minimum values likewise.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Sampling Functions

The functions described in this section manipulate signal samples. Sampling functions are used to change the sampling rate of the input signal and thus to obtain the signal vector of a required length. The functions perform the following operations:

- Insert zero-valued samples between neighboring samples of a signal (up-sample).
- Remove samples from between neighboring samples of a signal (down-sample).

The upsampling and downsampling functions are used by some filtering functions described in Chapter 6.

SampleUp

Up-samples a signal, conceptually increasing its sampling rate by an integer factor.

```
IppStatus ippsSampleUp_16s (const Ipp16s* pSrc, int srcLen,
     Ipp16s* pDst, int* pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleUp_32f (const Ipp32f* pSrc, int srcLen,
     Ipp32f* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_64f (const Ipp64f* pSrc, int srcLen,
     Ipp64f* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_16sc (const Ipp16sc* pSrc, int srcLen,
     Ipp16sc* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_32fc (const Ipp32fc* pSrc, int srcLen,
     Ipp32fc* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_64fc (const Ipp64fc* pSrc, int srcLen,
     Ipp64fc* pDst, int* pDstLen, int factor, int* pPhase);
```

Arguments

pSrc	Pointer to the input array (the signal to be up-sampled).
srcLen	Number of samples in the input array *pSrc*.
pDst	Pointer to the output array.
pDstLen	Pointer to the length value of the output array *pDst*.
factor	Factor by which the signal is up-sampled. That is, *factor* - 1 zeros are inserted after each sample of the input array *pSrc*.
pPhase	Pointer to the **input** phase value which determines where each sample from *pSrc* lies within each output block of *factor* samples in *pDst*. The value of *pPhase* is required to be in the range [0; *factor*-1]. The **output** value of *pPhase* can be used for the next up-sampling with the same *factor* and next *pSrc*.

Discussion

The function ippsSampleUp up-samples the *srcLen*-length input array *pSrc* by factor *factor* with phase *pPhase*, and stores the result in the array *pDst*, ignoring its length value by the *pDstLen* address.

Up-sampling inserts *factor*-1 zeros between each sample of *pSrc*. The *pPhase* argument determines where each sample from the input array lies within each output block of *factor* samples. The value of *pPhase* is required to be in the range [0; *factor*-1].

For example, if the input phase is 0, then every *factor* samples of the output array begin with the corresponding input array sample, the other *factor*-1 samples are equal to 0. The output array length is stored by the *pDstLen* address.

The *pPhase* value is the phase of an input array sample. It is also a returned output phase which can be used as an input phase for the first sample in the next block to process. Use *pPhase* for block mode processing to get a continuous output signal.

The ippsSampleUp functionality can be described as follows:

$pDst[factor * n + phase] = pSrc[n],\ 0 \le n < srcLen$

$pDst[factor * n + m] = 0,\ 0 \le n < srcLen, 0 \le m < factor,\ m \ne phase$

$pDstLen = factor * srcLen$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst*, *pSrc*, *pDstLen*, or *pPhase* pointer is NULL.
ippStsSizeErr	Indicates an error when *srcLen* is less than or equal to 0.
ippStsSampleFactorErr	Indicates an error when *factor* is less than or equal to 0.
ippStsSamplePhaseErr	Indicates an error when *pPhase* is negative, or bigger than or equal to *factor*.

SampleDown

Down-samples a signal, conceptually decreasing its sampling rate by an integer factor.

```
IppStatus ippsSampleDown_16s(const Ipp16s* pSrc, int srcLen,
      Ipp16s* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_32f(const Ipp32f* pSrc, int srcLen,
      Ipp32f* pDst, int* pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleDown_64f(const Ipp64f* pSrc, int srcLen,
     Ipp64f* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_32fc(const Ipp32fc* pSrc, int srcLen,
     Ipp32fc* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_64fc(const Ipp64fc* pSrc, int srcLen,
     Ipp64fc* Dst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_16sc(const Ipp16sc* pSrc, int srcLen,
     Ipp16sc* pDst, int* pDstLen, int factor, int* pPhase);
```

Arguments

pSrc	Pointer to the input array holding the samples to be down-sampled.
srcLen	Number of samples in the input array *pSrc*.
pDst	Pointer to the array that holds the output of the function ippsSampleDown.
pDstLen	Pointer to the length of the output array *pDst*.
factor	Factor by which the signal is down-sampled. That is, *factor* - 1 samples are discarded from every block of *factor* samples in *pSrc*.
pPhase	Pointer to the **input** phase value that determines which of the samples within each block of *factor* samples from *pSrc* is not discarded and copied to *pDst*. The value of *pPhase* is required to be in the range [0; *factor*-1]. The **output** value of *pPhase* can be used for the next down-sampling (sub-sampling) with the same *factor* and next *pSrc*.

Discussion

The function ippsSampleDown down-samples the *srcLen* -length array *pSrc* by factor *factor* with phase *pPhase*, and stores the result in the array *pDst*, ignoring its length value by the *pDstLen* address.

Down-sampling discards *factor* - 1 samples from *pSrc*, copying one sample from each block of *factor* samples from *pSrc* to *pDst*. The *pPhase* argument determines which of the samples in each block is not discarded and where it lies within each input block of *factor* samples. The value of *pPhase* is required to be in the range [0; *factor*-1]. The output array length is stored by the *pDstLen* address.

The *pPhase* value is the phase of an input array sample. It is also a returned output phase which can be used as an input phase for the first sample in the next block to process. Use *pPhase* for block mode processing to get a continuous output signal.

You can use the FIR multi-rate filter to combine filtering and resampling, for example, for antialiasing filtering before the sub-sampling procedure.

The ippsSampleDown functionality can be described as follows:

pDstLen = (srcLen + factor - 1 - phase)/factor

*pDst[n] = pSrc[factor * n + phase]*, $0 \le n < pDstLen$

phase = (factor + phase - srcLen%*factor)*%*factor*

Example 5-29 shows how to use the function ippsSampleDown.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst*, *pSrc*, *pDstLen*, or *pPhase* pointer is NULL.
ippStsSizeErr	Indicates an error when *srcLen* is less than or equal to 0.
ippStsSampleFactorErr	Indicates an error when *factor* is less than or equal to 0.
ippStsSamplePhaseErr	Indicates an error when *pPhase* is negative, or bigger than or equal to *factor*.

Example 5-29 Using the ippsSampleDown Function

```
void sampling( void ) {
      Ipp16s x[8] = { 1,2,3,4,5,6,7,8 };
      Ipp16s y[8] = {9,10,11,12,13,14,15,16}, z[8];
      int dstLen1, dstLen2, phase = 2;
      IppStatus st = ippsSampleDown_16s(x, 8, z, &dstLen1, 3, &phase);
      st = ippsSampleDown_16s(y, 8, z+dstLen1, &dstLen2, 3, &phase);
      printf_16s("down-sumpling =", z, dstLen1+dstLen2, st);
}
Output:
    down-sampling = 3 6 9 12 15
```

Filtering Functions 6

This chapter describes IPP functions that perform convolution and correlation operations, as well as linear and non-linear filtering.

Convolution and Correlation Functions

Convolution is an operation used to define an output signal from any linear time-invariant (LTI) processor in response to any input signal.

The correlation functions described later in this section estimate either the auto-correlation of a source vector or the cross-correlation of two vectors

Conv

Performs finite, linear convolution of two sequences.

```
IppStatus ippsConv_32f(const Ipp32f* pSrc1, int lenSrc1,
     const Ipp32f* pSrc2, int lenSrc2, Ipp32f* pDst);
IppStatus ippsConv_16s_Sfs(const Ipp16s* pSrc1, int lenSrc1,
     const Ipp16s* pSrc2, int lenSrc2, Ipp16s* pDst, int scaleFactor);
```

Arguments

pSrc1, pSrc2	Pointers to the two vectors to be convolved.
lenSrc1	Number of elements in the vector *pSrc1*.
lenSrc2	Number of elements in the vector *pSrc2*.

pDst	Pointer to the vector *pDst*. This vector stores the result of the convolution
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsConv performs finite linear convolution of two sequences. The *lenSrc1*- length vector *pSrc1* is convolved with the *lenSrc2*-length vector *pSrc2* to produce an (*lenSrc1* + *lenSrc2* - 1) -length vector *pDst*. The result of the convolution is defined as follows:

$$pDst[n] = \sum_{k=0}^{n} pSrc1[k] \cdot pSrc2[n-k], \quad 0 \le n < lenSrc1 + lenSrc2 - 1$$

Here *pSrc1[i]* = 0, if $i \ge lenSrc1$, and *pSrc2[j]* = 0, if $j \ge lenSrc2$.

Example 6-1 shows the code for the convolution of two vectors using ippsConv_16s_Sfs function.

Example 6-1 Using the ippsConv Function to Convolve Two Vectors

```
IppStatus convolution(void) {
      Ipp16s x[5] = {-2,0,1,-1,3}, h[2] = {0,1}, y[6];
      IppStatus st = ippsConv_16s_Sfs(x, 5, h, 2, y, 0);
      printf_16s("conv =", y, 6, st);
      return st;
}
Output:
    conv = 0 -2 0 1 -1 3
Matlab* Analog:
    >> x = [-2,0,1,-1,3]; h = [0,1]; y = conv(x,h)
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsMemAllocErr	Indicates an error when there is not enough memory for internal buffers.

ConvCyclic

Performs cyclic convolution of two sequences of the fixed size.

```
IppStatus ippsConvCyclic8x8_32f(const Ipp32f* x, const Ipp32f* h,
     Ipp32f* y);
IppStatus ippsConvCyclic4x4_32f32fc(const Ipp32f* x, const Ipp32fc* h,
     Ipp32fc* y);
IppStatus ippsConvCyclic8x8_16s_Sfs(const Ipp16s* x, const Ipp16s* h,
     Ipp16s* y, int scaleFactor);
```

Arguments

x, *h*	Pointers to the vectors to be convolved.
y	Pointer to the vector *y* that stores the result of convolution
scaleFactor	Refer to “Integer Scaling” in Chapter 2.

Discussion

The function ippsConvCyclic performs cyclic convolution of two sequences of the fixed size.

Return Value

ippStsNoErr	Indicates no error.

AutoCorr

Estimates normal, biased, and unbiased auto-correlation of a vector and stores the result in a second vector.

```
IppStatus ippsAutoCorr_32f(const Ipp32f* pSrc, int srcLen,
      Ipp32f* pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_32f(const Ipp32f* pSrc, int srcLen,
      Ipp32f* pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_32f(const Ipp32f* pSrc, int srcLen,
      Ipp32f* pDst, int dstLen);
IppStatus ippsAutoCorr_64f(const Ipp64f* pSrc, int srcLen,
      Ipp64f* pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_64f(const Ipp64f* pSrc, int srcLen,
      Ipp64f* pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_64f(const Ipp64f* pSrc, int srcLen,
      Ipp64f* pDst, int dstLen );
IppStatus ippsAutoCorr_32fc(const Ipp32fc* pSrc, int srcLen,
      Ipp32fc* pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_32fc(const Ipp32fc* pSrc, int srcLen,
      Ipp32fc* pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_32fc(const Ipp32fc* pSrc, int srcLen,
      Ipp32fc* pDst, int dstLen);
IppStatus ippsAutoCorr_64fc(const Ipp64fc* pSrc, int srcLen,
      Ipp64fc* pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_64fc(const Ipp64fc* pSrc, int srcLen,
      Ipp64fc* pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_64fc(const Ipp64fc* pSrc, int srcLen,
      Ipp64fc* pDst, int dstLen);
IppStatus ippsAutoCorr_16s_Sfs(const Ipp16s* pSrc, int srcLen,
      Ipp16s* pDst, int dstLen, int scaleFactor );
IppStatus ippsAutoCorr_NormA_16s_Sfs( const Ipp16s* pSrc, int srcLen,
      Ipp16s* pDst, int dstLen, int scaleFactor );
```

```
IppStatus ippsAutoCorr_NormB_16s_Sfs(const Ipp16s* pSrc, int srcLen,
     Ipp16s* pDst, int dstLen, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector
srcLen	The number of elements in the source vector.
pDst	Pointer to the destination vector, which stores the estimated auto-correlation results of the source vector.
dstLen	The number of elements in the destination vector (length of auto-correlation).
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The ippsAutoCorr function estimates normal auto-correlation of the *srcLen*-length source vector *pSrc* and stores the results in the *dstLen*-length vector *pDst*. Function flavors ippsAutoCorr_NormA and ippsAutoCorr_NormB compute biased and unbiased auto-correlation of the source vector, respectively. The resulting vector *pDst* is defined by the following equations:

$$pDst[n] = \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \le n < dstLen \qquad \text{(normal)}$$

$$pDst[n] = \frac{1}{srcLen} \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \le n < dstLen \qquad \text{(biased)}$$

$$pDst[n] = \frac{1}{srcLen-n} \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \le n < dstLen \qquad \text{(unbiased)}$$

where

$$pSrc[i] = \begin{cases} pSrc[i], & 0 \le i < srcLen \\ 0, & otherwise \end{cases}$$

Application Note: The auto-correlation estimates are computed only for positive lags, since the auto-correlation for a negative lag value is the complex conjugate of the auto-correlation for the equivalent positive lag.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *srcLen* or *dstLen* is less than or equal to 0.

See Also

`CrossCorr`	Estimates the cross-correlation of two vectors.

CrossCorr

Estimates the cross-correlation of two vectors.

```
IppStatus ippsCrossCorr_32f(const Ipp32f* pSrc1, int len1,
      const Ipp32f* pSrc2, int len2, Ipp32f* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_64f(const Ipp64f* pSrc1, int len1,
      const Ipp64f* pSrc2, int len2, Ipp64f* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_32fc(const Ipp32fc* pSrc1, int len1,
      const Ipp32fc* pSrc2, int len2, Ipp32fc* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_64fc(const Ipp64fc* pSrc1, int len1,
      const Ipp64fc* pSrc2, int len2, Ipp64fc* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_16s_Sfs(const Ipp16s* pSrc1, int len1,
      const Ipp16s* pSrc2, int len2, Ipp16s* pDst, int dstLen, int lowLag,
      int scaleFactor);
```

Arguments

pSrc1	Pointer to the first source vector.
len1	Number of elements in the vector *pSrc1*.
pSrc2	Pointer to the second source vector.
len2	Number of elements in the vector *pSrc2*.
pDst	Pointer to the vector which stores the results of the estimated cross-correlation of the vectors *pSrc1* and *pSrc2*.
dstLen	Number of elements in the vector *pDst*, which determines the range of lags at which the correlation estimates are computed.
lowLag	Lower value of the range of lags at which the correlation estimates are computed.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsCrossCorr estimates cross-correlation of the *len1*-length vector *pSrc1* and the *len2*-length vector *pSrc2*, and stores the results in the vector *pDst*. The resulting vector *pDst* is defined by the equation:

$$pDst[n] = \sum_{i=0}^{len1-1} conj(pSrc1[i]) \cdot pSrc2[n+i+lowLag],$$

where $0 \le n < dstLen$,

and

$$pSrc2[j] = \begin{cases} pSrc2[j], & 0 \le j < len2 \\ 0, & otherwise \end{cases}$$

Example 6-2 shows how to use the function ippsCrossCorr.

Example 6-2 Using the ippsCrossCorr Function

```
void crossCorr(void) {
      #undef LEN
      #define LEN 11
      Ipp32f win[LEN], y[LEN];
      IppStatus st;
      ippsSet_32f (1, win, LEN);
      ippsWinHamming_32f_I (win, LEN);
      st = ippsCrossCorr_32f (win, LEN, win, LEN, y, -(LEN-1));
      printf_32f("cross corr =", y,7,st);
}
Output:
    cross corr = 0.006400 0.026856 0.091831 0.242704 0.533230
    1.009000 1.672774
Matlab* analog:
    >> x = hamming(11)'; y = xcorr(x,x); y(1:7)
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrc1`* or *`pSrc2`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len1`* or *`len2`* is less than or equal to 0.

UpdateLinear

Integrates an input vector with specified integration weight.

```
IppStatus ippsUpdateLinear_16s32s_I(const Ipp16s* pSrc, int len,
     Ipp32s* pSrcDst, int srcShiftRight, Ipp16s alpha,
     IppHintAlgorithm hint);
```

Arguments

pSrc	Pointer to the source vector.
len	Number of elements of the vector.
pSrcDst	Pointer to the input value and output result.
srcShiftRight	Shift value; must be non-negative.
alpha	Integration weight.
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."

Discussion

The ippsUpdateLinear function performs *len* iterations in which the sum *alpha***pSrcDst* + (1-*alpha*)**pSrc*[*i*]$_{shift}$

is calculated and stored in *pSrcDst*.
Here *i* is the number of previous iterations, *pSrcDst* is the result of previous iteration, and *pSrc*[*i*]$_{shift}$ is a source vector element right-shifted by the non-negative value *srcShiftRight*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

UpdatePower

Integrates the square of an input vector with specified integration weight.

```
IppStatus ippsUpdatePower_16s32s_I(const Ipp16s* pSrc, int len,
     Ipp32s* pSrcDst, int srcShiftRight,Ipp16s alpha,
     IppHintAlgorithm hint);
```

Arguments

pSrc	Pointer to the source vector.
len	Number of elements of the vector.
pSrcDst	Pointer to input and output
srcShiftRight	Shift value.
alpha	Integration weight.
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."

Discussion

The ippsUpdatePower function performs *len* iterations in which the sum *alpha***pSrcDst* + (1-*alpha*)**pSrc*[*i*]$_{shift}$ **pSrc*[*i*]$_{shift}$

is calculated and stored in *pSrcDst*.
Here *i* is the number of previous iterations, *pSrcDst* is the result of previous iteration, and *pSrc*[*i*]$_{shift}$ is a source vector element right-shifted by the non-negative value *srcShiftRight*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Filtering Functions

The functions described in this section implement the following types of filters:

- finite impulse response (FIR) filter
- adaptive finite impulse response using least mean squares (LMS) filter
- infinite impulse response (IIR) filter
- median filter

FIR Filter Functions

The functions described in this section perform a finite impulse response filtering of input data. The functions initialize a finite impulse response filter, get and set the delay lines and filter coefficients (taps), and perform filtering. To use the FIR filter functions, follow this general scheme:

1. Call either `ippsFIRInitAlloc` to initialize the taps and the delay line in the state structure of a single-rate filter, or call `ippsFIRMRInitAlloc` to initialize the taps and the delay line in the state structure of a multi-rate filter.
2. Call `ippsFIROne` to filter a single sample through a single-rate filter and/or call `ippsFIR` to filter a block of consecutive samples through a single-rate or multi-rate filter.
3. Call `ippsFIRGetTaps` to get the filter coefficients (taps). Call `ippsFIRGetDlyLine` and `ippsFIRSetDlyLine` to get and set the values in the delay line.
4. Call `ippsFIRFree` to free dynamic memory associated with the FIR filter.

Alternatively, you may use the direct version of the functions. These functions perform filtering without initializing the filter state structure. All required parameters are directly set in the function.

FIRInitAlloc, FIRMRInitAlloc

Initializes a single-rate or multi-rate FIR filter state.

```
IppStatus ippsFIRInitAlloc_32f(IppsFIRState_32f** pState,
     const Ipp32f* pTaps, int tapsLen, const Ipp32f* pDlyLine);
IppStatus ippsFIRMRInitAlloc_32f(IppsFIRState_32f** pState,
     const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32f* pDlyLine);
IppStatus ippsFIRInitAlloc_64f(IppsFIRState_64f** pState,
     const Ipp64f* pTaps, int tapsLen, const Ipp64f* pDlyLine);
IppStatus ippsFIRMRInitAlloc_64f(IppsFIRState_64f** pState,
     const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp64f* pDlyLine);
IppStatus ippsFIRInitAlloc_32fc(IppsFIRState_32fc** pState,
     const Ipp32fc* pTaps, int tapsLen, const Ipp32fc* pDlyLine);
IppStatus ippsFIRMRInitAlloc_32fc(IppsFIRState_32fc** pState,
     const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32fc* pDlyLine);
IppStatus ippsFIRInitAlloc_64fc(IppsFIRState_64fc** pState,
     const Ipp64fc* pTaps, int tapsLen, const Ipp64fc* pDlyLine);
IppStatus ippsFIRMRInitAlloc_64fc(IppsFIRState_64fc** pState,
     const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp64fc* pDlyLine);


IppStatus ippsFIRInitAlloc32s_16s(IppsFIRState32s_16s** pState,
     const Ipp32s* pTaps, int tapsLen, int tapsFactor,
     const Ipp16s* pDlyLine);
IppStatus ippsFIRMRInitAlloc32s_16s(IppsFIRState32s_16s** pState,
     const Ipp32s* pTaps, int tapsLen, int tapsFactor, int upFactor,
     int upPhase, int downFactor, int downPhase, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc32sc_16sc(IppsFIRState32sc_16sc** pState,
     const Ipp32sc* pTaps, int tapsLen, int tapsFactor,
     const Ipp16sc* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc32sc_16sc(IppsFIRState32sc_16sc** pState,
     const Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor,
     int upPhase, int downFactor, int downPhase, const Ipp16sc* pDlyLine);

IppStatus ippsFIRInitAlloc32f_16s(IppsFIRState32f_16s** pState,
     const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine);
IppStatus ippsFIRMRInitAlloc32f_16s(IppsFIRState32f_16s** pState,
     const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc32fc_16sc(IppsFIRState32fc_16sc** pState,
     const Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc32fc_16sc(IppsFIRState32fc_16sc** pState,
     const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16sc* pDlyLine);

IppStatus ippsFIRInitAlloc64f_16s(IppsFIRState64f_16s** pState,
     const Ipp64f* pTaps, int tapsLen, const Ipp16s* pDlyLine);
IppStatus ippsFIRMRInitAlloc64f_16s(IppsFIRState64f_16s** pState,
     const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc64f_32s(IppsFIRState64f_32s** pState,
     const Ipp64f* pTaps, int tapsLen, const Ipp32s* pDlyLine);
IppStatus ippsFIRMRInitAlloc64f_32s(IppsFIRState64f_32s** pState,
     const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32s* pDlyLine);
IppStatus ippsFIRInitAlloc64f_32f(IppsFIRState64f_32f** pState,
     const Ipp64f* pTaps, int tapsLen, const Ipp32f* pDlyLine);
IppStatus ippsFIRMRInitAlloc64f_32f(IppsFIRState64f_32f** pState,
     const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32f* pDlyLine);

IppStatus ippsFIRInitAlloc64fc_16sc(IppsFIRState64fc_16sc** pState,
     const Ipp64fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc64fc_16sc(IppsFIRState64fc_16sc** pState,
     const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16sc* pDlyLine);
```

```
IppStatus ippsFIRInitAlloc64fc_32sc(IppsFIRState64fc_32sc** pState,
     const Ipp64fc* pTaps, int tapsLen, const Ipp32sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc64fc_32sc(IppsFIRState64fc_32sc** pState,
     const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32sc* pDlyLine);
IppStatus ippsFIRInitAlloc64fc_32fc(IppsFIRState64fc_32fc** pState,
     const Ipp64fc* pTaps, int tapsLen, const Ipp32fc* pDlyLine);
IppStatus ippsFIRMRInitAlloc64fc_32fc(IppsFIRState64fc_32fc** pState,
     const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32fc* pDlyLine);

IppStatus ippsFIRInitAlloc32s_16s32f(IppsFIRState32s_16s** pState,
     const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine);
IppStatus ippsFIRMRInitAlloc32s_16s32f(IppsFIRState32s_16s** pState,
     const Ipp32f *pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc32sc_16sc32fc(IppsFIRState32sc_16sc** pState,
     const Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc32sc_16sc32fc(IppsFIRState32sc_16sc**
     pState, const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16sc* pDlyLine);
```

Arguments

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of `Ipp32s` data type (for integer versions only).
downFactor	Factor used by the function `ippsFIRMRInit` for downsampling the multi-rate signals.
downPhase	Phase used by the function `ippsFIRMRInit` for downsampling the multi-rate signals.
upFactor	Factor used by the function `ippsFIRMRInit` for upsampling the multi-rate signals.

upPhase	Phase used by the function ippsFIRMRInit for upsampling the multi-rate signals.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *tapsLen* for single-rate filters and (*tapsLen* + *upFactor* - 1) / *upFactor* for multi-rate filters.
pState	Pointer to the FIR state structure to be created.

Discussion

The functions ippsFIRInitAlloc and ippsFIRMRInitAlloc create and initialize a single-rate or multi-rate FIR filter state, respectively. The initialization functions copy the taps from the *tapsLen*-length array *pTaps* into the state structure *pState*. To scale integer taps use the *tapsFactor* value. The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not NULL, the array content is copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.

If the state is not created, the initialization function returns an error status.

ippsFIRInitAlloc. The function ippsFIRInitAlloc initializes the taps and the delay line in the state structure *pState* of a single-rate filter. The *tapsLen*-length array *pTaps* specifies the taps. If the delay line array *pDlyLine* is non-NULL its length is equal to *tapsLen*.

ippsFIRMRInitAlloc. The function ippsFIRMRInitAlloc initializes the taps and the delay line in the state structure *pState* of a multi-rate filter; that is, a filter that internally upsamples and/or downsamples using a polyphase filter structure. It initializes the state structure in the same way as described for single-rate filters, but includes additional information about the required upsampling and downsampling parameters.

The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see ippsSampleUp function on page 5-121). That is, *upFactor*-1 zeros are inserted between each sample of input signal.

The argument *upPhase* is the parameter which determines where a non-zero sample lies within the *upFactor*-length block of upsampled input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal is internally downsampled (see ippsSampleDown function on page 5-123). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of upsampled filter response.

The argument *downPhase* is the parameter which determines where non-discarded sample lies within a block of upsampled filter response.

If the delay line array *pDelay* is non-NULL its length is defined as (*tapsLen* + *upFactor* - 1) / *upFactor*.

Return Value

ippStsNoErr	Indicates no error.
ippStsMemAllocErr	Indicates an error when no memory is allocated.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL .
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than or equal to 0.
ippStsFIRMRFactorErr	Indicates an error when *upFactor* (*downFactor*) is less than or equal to 0.
ippStsFIRMRPhaseErr	Indicates an error when *upPhase* (*downPhase*) is negative, or less than or equal to *upFactor* (*downFactor*).
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRFree

Closes a FIR filter state.

```
IppStatus ippsFIRFree_32f(IppsFIRState_32f* pState);
IppStatus ippsFIRFree_64f(IppsFIRState_64f* pState);
IppStatus ippsFIRFree_32fc(IppsFIRState_32fc* pState);
IppStatus ippsFIRFree_64fc(IppsFIRState_64fc* pState);
IppStatus ippsFIRFree32s_16s(IppsFIRState32s_16s* pState);
IppStatus ippsFIRFree32f_16s(IppsFIRState32f_16s* pState);
```

```
IppStatus ippsFIRFree32sc_16sc(IppsFIRState32sc_16sc* pState);
IppStatus ippsFIRFree32fc_16sc(IppsFIRState32fc_16sc* pState);
IppStatus ippsFIRFree64f_16s(IppsFIRState64f_16s* pState);
IppStatus ippsFIRFree64f_32s(IppsFIRState64f_32s* pState);
IppStatus ippsFIRFree64f_32f(IppsFIRState64f_32f* pState);
IppStatus ippsFIRFree64fc_16sc(IppsFIRState64fc_16sc* pState);
IppStatus ippsFIRFree64fc_32sc(IppsFIRState64fc_32sc* pState);
IppStatus ippsFIRFree64fc_32fc(IppsFIRState64fc_32fc* pState);
```

Arguments

pState	Pointer to the FIR filter state structure to be closed.

Discussion

The function ippsFIRFree closes the FIR filter state by freeing all memory associated with the filter state created by ippsFIRInitAlloc or ippsFIRMRInitAlloc. Call ippsFIRFree after filtering is completed.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL .
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIROne

Filters a single sample through a FIR filter.

```
IppStatus ippsFIROne_32f(Ipp32f src, Ipp32f* pDstVal,
     IppsFIRState_32f* pState);
IppStatus ippsFIROne_64f(Ipp64f src, Ipp64f* pDstVal,
     IppsFIRState_64f* pState);
IppStatus ippsFIROne64f_32f(Ipp32f src, Ipp32f* pDstVal,
     IppsFIRState64f_32f* pState);
```

```
IppStatus ippsFIROne_32fc(Ipp32fc src, Ipp32fc* pDstVal,
     IppsFIRState_32fc* pState);
IppStatus ippsFIROne_64fc(Ipp64fc src, Ipp64fc* pDstVal,
     IppsFIRState_64fc* pState);
IppStatus ippsFIROne64fc_32fc(Ipp32fc src, Ipp32fc* pDstVal,
     IppsFIRState64fc_32fc* pState);

IppStatus ippsFIROne32s_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
     IppsFIRState32s_16s* pState, int scaleFactor);
IppStatus ippsFIROne32f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
     IppsFIRState32f_16s* pState, int scaleFactor);
IppStatus ippsFIROne64f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
     IppsFIRState64f_16s* pState, int scaleFactor);
IppStatus ippsFIROne64f_32s_Sfs(Ipp32s src, Ipp32s* pDstVal,
     IppsFIRState64f_32s* pState, int scaleFactor);

IppStatus ippsFIROne32sc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
     IppsFIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne32fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
     IppsFIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne64fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
     IppsFIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne64fc_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
     IppsFIRState64fc_32sc* pState, int scaleFactor);
```

Arguments

pState	Pointer to the FIR filter state structure.
src	Input sample to be filtered by the function ippsFIROne.
pDstVal	Pointer to the output sample filtered by the function ippsFIROne.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsFIROne filters a single sample *src* through a single-rate filter and stores the result in *pDstVal*. The filter parameters are specified in *pState*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated. In the following definition of the FIR filter, the sample to be filtered is denoted *x(n)* and the taps are denoted *h(i)*.

The return value *y(n)* is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

Before calling ippsFIROne, initialize the filter state by calling ippsFIRInitAlloc. Specify the number of taps *tapsLen*, the tap values in *pTaps*, and the delay line values in *pDelay* earlier.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIROne_Direct

Directly filters a single sample through a FIR filter.

```
IppStatus ippsFIROne_Direct_32f(Ipp32f src, Ipp32f* pDstVal,
     const Ipp32f* pTaps, int tapsLen, Ipp32f* pDlyLine,
     int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_64f(Ipp64f src, Ipp64f* pDstVal,
     const Ipp64f* pTaps, int tapsLen, Ipp64f* pDlyLine,
     int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_32fc(Ipp32fc src, Ipp32fc* pDstVal,
     const Ipp32fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
     int* pDlyLineIndex);
```

```
IppStatus ippsFIROne_Direct_64fc(Ipp64fc src, Ipp64fc* pDstVal,
      const Ipp64fc* pTaps, int tapsLen, Ipp64fc* pDlyLine,
      int* pDlyLineIndex);

IppStatus ippsFIROne64f_Direct_32f(Ipp32f src, Ipp32f* pDstVal,
      const Ipp64f* pTaps, int tapsLen, Ipp32f* pDlyLine,
      int* pDlyLineIndex);

IppStatus ippsFIROne64fc_Direct_32fc(Ipp32fc src, Ipp32fc* pDstVal,
      const Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
      int* pDlyLineIndex);


IppStatus ippsFIROne32f_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
      const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64f_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
      const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64f_Direct_32s_Sfs(Ipp32s src, Ipp32s* pDstVal,
      const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne32fc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
      const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64fc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
      const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64fc_Direct_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
      const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);


IppStatus ippsFIROne32s_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
      const Ipp32s* pTaps, int tapsLen, int tapsFactor,
      Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne32sc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
      const Ipp32sc* pTaps, int tapsLen, int tapsFactor,
      Ipp16sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIROne_Direct_32f_I(Ipp32f* pSrcDstVal,
     const Ipp32f* pTaps, int tapsLen, Ipp32f* pDlyLine,
     int* pDlyLineIndex);

IppStatus ippsFIROne_Direct_64f_I(Ipp64f* pSrcDstVal,
     const Ipp64f* pTaps, int tapsLen, Ipp64f* pDlyLine,
     int* pDlyLineIndex);

IppStatus ippsFIROne_Direct_32fc_I(Ipp32fc* pSrcDstVal,
     const Ipp32fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
     int* pDlyLineIndex);

IppStatus ippsFIROne_Direct_64fc_I(Ipp64fc* pSrcDstVal,
     const Ipp64fc* pTaps, int tapsLen, Ipp64fc* pDlyLine,
     int* pDlyLineIndex);

IppStatus ippsFIROne64f_Direct_32f_I(Ipp32f* pSrcDstVal,
     const Ipp64f* pTaps, int tapsLen, Ipp32f* pDlyLine,
     int* pDlyLineIndex);

IppStatus ippsFIROne64fc_Direct_32fc_I(Ipp32fc* pSrcDstVal,
     const Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
     int* pDlyLineIndex);


IppStatus ippsFIROne32f_Direct_16s_ISfs(Ipp16s* pSrcDstVal,
     const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64f_Direct_16s_ISfs(Ipp16s* pSrcDstVal,
     const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64f_Direct_32s_ISfs(Ipp32s* pSrcDstVal,
     const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal,
     const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal,
     const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDstVal,
     const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIROne32s_Direct_16s_ISfs(Ipp16s* pSrcDstVal,
     const Ipp32s* pTaps, int tapsLen, int tapsFactor,
     Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal,
     const Ipp32sc* pTaps, int tapsLen, int tapsFactor,
     Ipp16sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);
```

Arguments

src	Input sample to be filtered by the function.
pDstVal	Pointer to the output sample filtered by the function.
pSrcDstVal	Pointer to the input and output sample for in-place operation.
pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of Ipp32s data type (for integer versions only).
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *2*tapsLen* .
pDlyLineIndex	Pointer to the current delay line index.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The ippsFIROne_Direct function directly filters a single sample *src* or *pSrcDstVal* through a single-rate filter and stores the result in *pDstVal* or *pSrcDstVal*. The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps, the *tapsFactor* value is used. The set of *tapsLen* input samples is copied twice to the *2*tapsLen*-length array *pDlyLine*. The current delay line index is specified in the *pDlyLineIndex*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated. In the following definition of the FIR filter, the sample to be filtered is denoted *x(n)* and the taps are denoted *h(i)*.

The return value *y(n)* is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL .
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than or equal to 0.

FIR

Filters a block of samples through a single-rate or multi-rate FIR filter.

```
IppStatus ippsFIR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int numIters, IppsFIRState_32f* pState);
IppStatus ippsFIR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
      int numIters, IppsFIRState_64f* pState);
IppStatus ippsFIR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      int numIters, IppsFIRState_32fc* pState);
IppStatus ippsFIR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
      int numIters, IppsFIRState_64fc* pState);
IppStatus ippsFIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int numIters, IppsFIRState64f_32f* pState);
IppStatus ippsFIR64fc_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      int numIters, IppsFIRState64fc_32fc* pState);

IppStatus ippsFIR32s_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
      int numIters, IppsFIRState32s_16s* pState, int scaleFactor);
IppStatus ippsFIR32f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
      int numIters, IppsFIRState32f_16s* pState, int scaleFactor);
```

```
IppStatus ippsFIR64f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     int numIters, IppsFIRState64f_16s* pState, int scaleFactor);

IppStatus ippsFIR64f_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst,
     int numIters, IppsFIRState64f_32s* pState, int scaleFactor);

IppStatus ippsFIR32sc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int numIters, IppsFIRState32sc_16sc* pState, int scaleFactor);

IppStatus ippsFIR32fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int numIters, IppsFIRState32fc_16sc* pState, int scaleFactor);

IppStatus ippsFIR64fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int numIters, IppsFIRState64fc_16sc* pState, int scaleFactor);

IppStatus ippsFIR64fc_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst,
     int numIters, IppsFIRState64fc_32sc* pState, int scaleFactor);


IppStatus ippsFIR_32f_I(Ipp32f* pSrcDst, int numIters,
     IppsFIRState_32f* pState);

IppStatus ippsFIR_64f_I(Ipp64f* pSrcDst, int numIters,
     IppsFIRState_64f* pState);

IppStatus ippsFIR64f_32f_I(Ipp32f* pSrcDst, int numIters,
     IppsFIRState64f_32f* pState);

IppStatus ippsFIR_32fc_I(Ipp32fc* pSrcDst, int numIters,
     IppsFIRState_32fc* pState);

IppStatus ippsFIR_64fc_I(Ipp64fc* pSrcDst, int numIters,
     IppsFIRState_64fc* pState);

IppStatus ippsFIR64fc_32fc_I(Ipp32fc* pSrcDst, int numIters,
     IppsFIRState64fc_32fc* pState);


IppStatus ippsFIR32s_16s_ISfs(Ipp16s* pSrcDst, int numIters,
     IppsFIRState32s_16s* pState, int scaleFactor);

IppStatus ippsFIR32f_16s_ISfs(Ipp16s* pSrcDst, int numIters,
     IppsFIRState32f_16s* pState, int scaleFactor);

IppStatus ippsFIR64f_16s_ISfs(Ipp16s* pSrcDst, int numIters,
     IppsFIRState64f_16s* pState, int scaleFactor);

IppStatus ippsFIR64f_32s_ISfs(Ipp32s* pSrcDst, int numIters,
     IppsFIRState64f_32s* pState, int scaleFactor);

IppStatus ippsFIR32sc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
     IppsFIRState32sc_16sc* pState, int scaleFactor);
```

```
IppStatus ippsFIR32fc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
     IppsFIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsFIR64fc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
     IppsFIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsFIR64fc_32sc_ISfs(Ipp32sc* pSrcDst, int numIters,
     IppsFIRState64fc_32sc* pState, int scaleFactor);
```

Arguments

pState	Pointer to the FIR filter state structure.
pSrc	Pointer to the input array to be filtered by the function ippsFIR.
pDst	Pointer to the output array filtered by the function ippsFIR.
pSrcDst	Pointer to the input and output array (for the in-place operation) to be filtered by the function ippsFIR.
numIters	Parameter associated with the number of samples to be filtered by the function ippsFIR. For single-rate filters, the *numIters* samples in the input array are filtered and the resulting *numIters* samples are stored in the output array. For multi-rate filters, the (*numIters* * *downFactor*) samples in the input array are filtered and the resulting (*numIters* * *upFactor*) samples are stored in the output array.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsFIR filters an input array *pSrc* or *pSrcDst* through a single-rate or multi-rate filter, and stores the results in *pDst* or *pSrcDst*, respectively. The filter parameters are specified in *pState*.

For single-rate filters, the *numIters* samples in the array *pSrc* or *pSrcDst* are filtered, and the resulting *numIters* samples are stored in the array *pDst* or *pSrcDst*. The results are identical to *numIters* consecutive calls to ippsFIROne.

In the following definition of the FIR filter, the sample to be filtered is denoted *x(n)*, the taps are denoted *h(i)*, and the return value is *y(n)*.

The return value *y(n)* is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i), \quad 0 \le n < numIters$$

After the function has performed calculations, it updates the delay line values stored in the state. For a single-rate filter the *numIters* parameter defines the length of the source and destination arrays.

For multi-rate filters, ippsFIR filters (*numIters***downFactor*) input samples and stores the resulting (*numIters***upFactor*) samples in the output array. The multi-rate filtering is considered as a sequence of three operations: upsampling, filtering with a single-rate FIR filter, and downsampling. The algorithm is implemented as a single operation including the above-mentioned three steps. Thus, the function does not create an internal (*upFactor***srcLen*)-size buffer to store the upsampling result.

Before calling ippsFIR, initialize the filter state by calling either ippsFIRInitAlloc or ippsFIRMRInitAlloc. Specify the number of taps *tapsLen*, the tap values in *pTaps*, and the delay line values in *pDelay* beforehand.

Example 6-3 illustrates single-rate filtering with the function ippsFIR_32f.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsSizeErr	Indicates an error when *numIters* is less or equal to 0.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

Example 6-3 Single-Rate Filtering with the ippsFIR Function

```
IppStatus fir(void) {
#undef NUMITERS
#define NUMITERS 150
     int n;
     IppStatus status;
     IppsIIRState_32f *ictx;
     IppsFIRState_32f *fctx;
     Ipp32f *x = ippsMalloc_32f(NUMITERS),
            *y = ippsMalloc_32f(NUMITERS),
            *z = ippsMalloc_32f(NUMITERS);
     const float taps[] = {
         0.0051f, 0.0180f, 0.0591f, 0.1245f, 0.1869f, 0.2127f, 0.1869f,
         0.1245f, 0.0591f, 0.0180f, 0.0051f, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0
     };
     for (n =0;n<NUMITERS;++n)x[n]=(float)sin(IPP_2PI *n *0.2);
     ippsIIRInitAlloc_32f( &ictx, taps, 10, NULL );
     ippsFIRInitAlloc_32f( &fctx, taps, 11, NULL );
     status = ippsIIR_32f( x, y, NUMITERS, ictx);
     printf_32f("IIR 32f output + 120 =", y+120, 5, status);
     ippsIIRFree_32f(ictx);
     status = ippsFIR_32f( x, z, NUMITERS, fctx );
     printf_32f("FIR 32f output + 120 =", z+120, 5, status);
     ippsFIRFree_32f(fctx);
     ippsFree(z);
     ippsFree(y);
     ippsFree(x);
     return status;
}
Output:
    IIR 32f output + 120 = 0.000000 0.049896 0.030838 -0.030838 -0.049896
    FIR 32f output + 120 = 0.000000 0.049896 0.030838 -0.030838 -0.049896
Matlab* Analog:
    >> F = 0.2; N = 150; n = 0:N-1; x = sin(2*pi*n*F);
    y = filter(fir1(10,0.15),1,x); y(121:125)
```

FIR_Direct

Directly filters a block of samples through a single-rate FIR filter.

```
IppStatus ippsFIR_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int numIters, const Ipp32f* pTaps, int tapsLen,
      Ipp32f* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR_Direct_64f(const Ipp64f* pSrc, Ipp64f* pDst,
      int numIters, const Ipp64f* pTaps, int tapsLen,
      Ipp64f* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      int numIters, const Ipp32fc* pTaps, int tapsLen,
      Ipp32fc* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR_Direct_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
      int numIters, const Ipp64fc* pTaps, int tapsLen,
      Ipp64fc* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR64f_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int numIters, const Ipp64f* pTaps, int tapsLen,
      Ipp32f* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR64fc_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      int numIters, const Ipp64fc* pTaps, int tapsLen,
      Ipp32fc* pDlyLine, int* pDlyLineIndex);


IppStatus ippsFIR32f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
      int numIters, const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR64f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
      int numIters, const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR64f_Direct_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst,
      int numIters, const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR32fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
      int numIters, const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR64fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
      int numIters, const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64fc_Direct_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst,
      int numIters, const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR32s_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
      int numIters, const Ipp32s* pTaps, int tapsLen, int tapsFactor,
      Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32sc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
      int numIters, const Ipp32sc* pTaps, int tapsLen, int tapsFactor,
      Ipp16sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR_Direct_32f_I(Ipp32f* pSrcDst, int numIters,
      const Ipp32f* pTaps, int tapsLen, Ipp32f* pDlyLine,
      int* pDlyLineIndex);
IppStatus ippsFIR_Direct_64f_I(Ipp64f* pSrcDst, int numIters,
      const Ipp64f* pTaps, int tapsLen, Ipp64f* pDlyLine,
      int* pDlyLineIndex);
IppStatus ippsFIR_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters,
      const Ipp32fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
      int* pDlyLineIndex);
IppStatus ippsFIR_Direct_64fc_I(Ipp64fc* pSrcDst, int numIters,
      const Ipp64fc* pTaps, int tapsLen, Ipp64fc* pDlyLine,
      int* pDlyLineIndex);
IppStatus ippsFIR64f_Direct_32f_I(Ipp32f* pSrcDst, int numIters,
      const Ipp64f* pTaps, int tapsLen, Ipp32f* pDlyLine,
      int* pDlyLineIndex);
IppStatus ippsFIR64fc_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters,
      const Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
      int* pDlyLineIndex);

IppStatus ippsFIR32f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
      const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR64f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
     const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64f_Direct_32s_ISfs(Ipp32s* pSrcDst, int numIters,
     const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
     const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
     const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDst, int numIters,
     const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR32s_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
     const Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
     const Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);
```

Arguments

pSrc	Pointer to the input array to be filtered.
pDst	Pointer to the output array.
pSrcDst	Pointer to the input and output array for the in-place operation.
numIters	Number of samples in the input array to be filtered.
pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of `Ipp32s` data type (for integer versions only).

pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *2*tapsLen*.
pDlyLineIndex	Pointer to the current delay line index.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsFIR_Direct filters an input array *pSrc* or *pSrcDst* containing *numIters* samples through a single-rate filter, and stores the resulting *numIters* samples in *pDst* or *pSrcDst*, respectively. The results are identical to *numIters* consecutive calls to ippsFIROne_Direct.
The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps the *tapsFactor* value is used. The set of *tapsLen* input samples is copied to the *2*tapsLen*-length array *pDlyLine*. The current delay line index is specified in the *pDlyLineIndex*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

In the following definition of the FIR filter, the sample to be filtered is denoted *x(n)*, the taps are denoted *h(i)*, and the return value is *y(n)*.

The return value *y(n)* is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i), \quad 0 \le n < numIters$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL .
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than or equal to 0.
ippStsSizeErr	Indicates an error when *numIters* is less than or equal to 0.

FIRMR_Direct

Directly filters a block of samples through a multi-rate FIR filter.

```
IppStatus ippsFIRMR_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int numIters, const Ipp32f* pTaps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase, Ipp32f* pDlyLine);

IppStatus ippsFIRMR_Direct_64f(const Ipp64f* pSrc, Ipp64f* pDst,
      int numIters, const Ipp64f* pTaps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase, Ipp64f* pDlyLine);

IppStatus ippsFIRMR_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      int numIters, const Ipp32fc* pTaps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase, Ipp32fc* pDlyLine);

IppStatus ippsFIRMR_Direct_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
      int numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase, Ipp64fc* pDlyLine);

IppStatus ippsFIRMR64f_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int numIters, const Ipp64f* pTaps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase, Ipp32f* pDlyLine);

IppStatus ippsFIRMR64fc_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      int numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase, Ipp32fc* pDlyLine);


IppStatus ippsFIRMR32f_Direct_16s_Sfs(const Ipp16s* pSrc,
      Ipp16s* pDst, int numIters, const Ipp32f* pTaps, int tapsLen,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp16s* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64f_Direct_16s_Sfs(const Ipp16s* pSrc,
      Ipp16s* pDst, int numIters, const Ipp64f* pTaps, int tapsLen,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp16s* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64f_Direct_32s_Sfs(const Ipp32s* pSrc,
      Ipp32s* pDst, int numIters, const Ipp64f* pTaps, int tapsLen,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp32s* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR32fc_Direct_16sc_Sfs(const Ipp16sc* pSrc,
      Ipp16sc* pDst, int numIters, const Ipp32fc* pTaps, int tapsLen,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp16sc* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64fc_Direct_16sc_Sfs(const Ipp16sc* pSrc,
      Ipp16sc* pDst, int numIters, const Ipp64fc* pTaps, int tapsLen,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp16sc* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64fc_Direct_32sc_Sfs(const Ipp32sc* pSrc,
      Ipp32sc* pDst, int numIters, const Ipp64fc* pTaps, int tapsLen,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp32sc* pDlyLine, int scaleFactor);


IppStatus ippsFIRMR32s_Direct_16s_Sfs(const Ipp16s* pSrc,
      Ipp16s* pDst, int numIters, const Ipp32s* pTaps, int tapsLen,
      int tapsFactor, int upFactor, int upPhase, int downFactor,
      int downPhase, Ipp16s* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR32sc_Direct_16sc_Sfs(const Ipp16sc* pSrc,
      Ipp16sc* pDst, int numIters, const Ipp32sc* pTaps, int tapsLen,
      int tapsFactor, int upFactor, int upPhase, int downFactor,
      int downPhase, Ipp16sc* pDlyLine, int scaleFactor)


IppStatus ippsFIRMR_Direct_32f_I(Ipp32f* pSrcDst, int numIters,
      const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp32f* pDlyLine);

IppStatus ippsFIRMR_Direct_64f_I(Ipp64f* pSrcDst, int numIters,
      const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp64f* pDlyLine);;)

IppStatus ippsFIRMR_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters,
      const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp32fc* pDlyLine);

IppStatus ippsFIRMR_Direct_64fc_I(Ipp64fc* pSrcDst, int numIters,
      const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp64fc* pDlyLine);

IppStatus ippsFIRMR64f_Direct_32f_I(Ipp32f* pSrcDst, int numIters,
      const Ipp64f* pTaps, int tapsLen, int upFactor,int upPhase,
      int downFactor, int downPhase, Ipp32f* pDlyLine);
```

```
IppStatus ippsFIRMR64fc_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters,
      const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp32fc* pDlyLine);


IppStatus ippsFIRMR32f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
      const Ipp32f* pTaps, int tapsLen, int upFactor,int upPhase,
      int downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
      const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64f_Direct_32s_ISfs(Ipp32s* pSrcDst, int numIters,
      const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp32s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
      const Ipp32fc* pTaps, int tapsLen, int upFactor,int upPhase,
      int downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
      const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDst, int numIters,
      const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp32sc* pDlyLine, int scaleFactor);


IppStatus ippsFIRMR32s_Direct_16s_ISfs(Ipp16s* pSrcDst,
      int numIters, const Ipp32s* pTaps, int tapsLen, int tapsFactor,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDst,
      int numIters, const Ipp32sc* pTaps, int tapsLen, int tapsFactor,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp16sc* pDlyLine, int scaleFactor);
```

Arguments

pSrc	Pointer to the input array to be filtered.
pDst	Pointer to the output array.
pSrcDst	Pointer to the input and output array for the in-place operation.

numIters	Parameter associated with the number of samples to be filtered by the function. The (*numIters* * *downFactor*) samples of the input array are filtered and the resulting (*numIters* * *upFactor*) samples are stored in the output array
pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of Ipp32s data type (for integer versions only).
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is (*tapsLen* + *upFactor* - 1) / *upFactor*.
upFactor	Factor for upsampling the multi-rate signals.
downFactor	Factor for downsampling the multi-rate signals.
upPhase	Phase for upsampling the multi-rate signals.
downPhase	Phase for downsampling the multi-rate signals.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsFIRMR_Direct filters an input array *pSrc* or *pSrcDst* through a multi-rate filter, and stores the resulting samples in *pDst* or *pSrcDst*, respectively. The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps the *tapsFactor* value is used. The array *pDlyLine* specifies the delay line values. The input array contains (*numIters***downFactor*) samples, and the output array stores the resulting (*numIters***upFactor*) samples.
The multi-rate filtering is considered as a sequence of three operations: upsampling, filtering with a single-rate FIR filter, and downsampling. The algorithm is implemented as a single operation including the above-mentioned three steps.

The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see ippsSampleUp function on page 5-121). That is, *upFactor*-1 zeros are inserted between each sample of input signal.

The argument *upPhase* is the parameter which determines where a non-zero sample lies within the *upFactor*-length block of upsampled input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal is internally downsampled (see ippsSampleDown function on page 5-123). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of upsampled filter response.

The argument *downPhase* is the parameter which determines where non-discarded sample lies within a block of upsampled filter response. The length of the delay line array *pDelay* is defined as (*tapsLen* + *upFactor* - 1) / *upFactor*.
The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than or equal to 0.
ippStsSizeErr	Indicates an error when *numIters* is less than or equal to 0.
ippStsFIRMRFactorErr	Indicates an error when *upFactor* (*downFactor*) is less than or equal to 0.
ippStsFIRMRPhaseErr	Indicates an error when *upPhase* (*downPhase*) is negative, or less than or equal to *upFactor* (*downFactor*).

FIRGetTaps

Gets the taps of a FIR filter state.

```
IppStatus ippsFIRGetTaps_32f(const IppsFIRState_32f* pState,
     Ipp32f* pTaps);
IppStatus ippsFIRGetTaps_64f(const IppsFIRState_64f* pState,
     Ipp64f* pTaps);
```

```
IppStatus ippsFIRGetTaps32f_16s(const IppsFIRState32f_16s* pState,
     Ipp32f* pTaps);
IppStatus ippsFIRGetTaps64f_16s(const IppsFIRState64f_16s* pState,
     Ipp64f* pTaps);
IppStatus ippsFIRGetTaps64f_32s(const IppsFIRState64f_32s* pState,
     Ipp64f* pTaps);
IppStatus ippsFIRGetTaps64f_32f(const IppsFIRState64f_32f* pState,
     Ipp64f* pTaps);
IppStatus ippsFIRGetTaps_32fc(const IppsFIRState_32fc* pState,
     Ipp32fc* pTaps);
IppStatus ippsFIRGetTaps_64fc(const IppsFIRState_64fc* pState,
     Ipp64fc* pTaps);
IppStatus ippsFIRGetTaps32fc_16sc(const IppsFIRState32fc_16sc* pState,
     Ipp32fc* pTaps);
IppStatus ippsFIRGetTaps64fc_16sc(const IppsFIRState64fc_16sc* pState,
     Ipp64fc* pTaps);
IppStatus ippsFIRGetTaps64fc_32sc(const IppsFIRState64fc_32sc* pState,
     Ipp64fc* pTaps);
IppStatus ippsFIRGetTaps64fc_32fc(const IppsFIRState64fc_32fc* pState,
     Ipp64fc* pTaps);
IppStatus ippsFIRGetTaps32s_16s32f(const IppsFIRState32s_16s* pState,
     Ipp32f* pTaps);
IppStatus ippsFIRGetTaps32sc_16sc32fc(const IppsFIRState32sc_16sc*
     pState, Ipp32fc* pTaps);

IppStatus ippsFIRGetTaps32s_16s(const IppsFIRState32s_16s* pState,
     Ipp32s* pTaps, int* tapsFactor);
IppStatus ippsFIRGetTaps32sc_16sc(const IppsFIRState32sc_16sc* pState,
     Ipp32sc* pTaps, int* tapsFactor);
```

Arguments

pState	Pointer to the FIR filter state structure.
pTaps	Pointer to the array holding copies of the taps.
tapsFactor	Scale factor for the taps of `Ipp32s` data type (for integer versions only).

Discussion

The function ippsFIRGetTaps copies the taps from the state structure *pState* to the *tapsLen*-length array *pTaps*. To scale integer taps use the *tapsFactor* value. To set new taps in the state structure, create a new state using the function ippsFIRInit or ippsFIRMRInit.

Before calling the function ippsFIRGetTaps, initialize the filter state by calling ippspFIRInitAlloc or ippspFIRMRInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRGetDlyLine, FIRSetDlyLine

Gets and sets the delay line contents of a FIR filter state.

```
IppStatus ippsFIRGetDlyLine_32f(const IppsFIRState_32f* pState,
      Ipp32f* pDlyLine);
IppStatus ippsFIRGetDlyLine_64f(const IppsFIRState_64f* pState,
      Ipp64f* pDlyLine);
IppStatus, ippsFIRGetDlyLine32s_16s, (const IppsFIRState32s_16s* pState,
      Ipp16s* pDlyLine);
IppStatus ippsFIRGetDlyLine32f_16s(const IppsFIRState32f_16s* pState,
      Ipp16s* pDlyLine);
IppStatus ippsFIRGetDlyLine64f_16s(const IppsFIRState64f_16s* pState,
      Ipp16s* pDlyLine);
IppStatus ippsFIRGetDlyLine64f_32s(const IppsFIRState64f_32s* pState,
      Ipp32s* pDlyLine);
IppStatus ippsFIRGetDlyLine64f_32f(const IppsFIRState64f_32f* pState,
      Ipp32f* pDlyLine);
```

```
IppStatus ippsFIRGetDlyLine_32fc(const IppsFIRState_32fc* pState,
     Ipp32fc* pDlyLine);
IppStatus ippsFIRGetDlyLine_64fc(const IppsFIRState_64fc* pState,
     Ipp64fc* pDlyLine);
IppStatus, ippsFIRGetDlyLine32sc_16sc, (const IppsFIRState32sc_16sc*
     pState, Ipp16sc* pDlyLine);
IppStatus ippsFIRGetDlyLine32fc_16sc(const IppsFIRState32fc_16sc*
     pState, Ipp16sc* pDlyLine);
IppStatus ippsFIRGetDlyLine64fc_16sc(const IppsFIRState64fc_16sc*
     pState, Ipp16sc* pDlyLine);
IppStatus ippsFIRGetDlyLine64fc_32sc(const IppsFIRState64fc_32sc*
     pState, Ipp32sc* pDlyLine);
IppStatus ippsFIRGetDlyLine64fc_32fc(const IppsFIRState64fc_32fc*
     pState, Ipp32fc* pDlyLine);


IppStatus ippsFIRSetDlyLine_32f(IppsFIRState_32f* pState,
     const Ipp32f* pDlyLine);
IppStatus ippsFIRSetDlyLine_64f(IppsFIRState_64f* pState,
     const Ipp64f* pDlyLine);
IppStatus, ippsFIRSetDlyLine32s_16s, (IppsFIRState32s_16s* pState,
     const Ipp16s* pDlyLine);
IppStatus ippsFIRSetDlyLine32f_16s(IppsFIRState32f_16s* pState,
     const Ipp16s* pDlyLine);
IppStatus ippsFIRSetDlyLine64f_16s(IppsFIRState64f_16s* pState,
     const Ipp16s* pDlyLine);
IppStatus ippsFIRSetDlyLine64f_32s(IppsFIRState64f_32s* pState,
     const Ipp32s* pDlyLine);
IppStatus ippsFIRSetDlyLine64f_32f(IppsFIRState64f_32f* pState,
     const Ipp32f* pDlyLine);


IppStatus ippsFIRSetDlyLine_32fc(IppsFIRState_32fc* pState,
     const Ipp32fc* pDlyLine);
IppStatus ippsFIRSetDlyLine_64fc(IppsFIRState_64fc* pState,
     const Ipp64fc* pDlyLine);
```

```
IppStatus, ippsFIRSetDlyLine32sc_16sc, (IppsFIRState32sc_16sc* pState,
     const Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine32fc_16sc(IppsFIRState32fc_16sc* pState,
     const Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_16sc(IppsFIRState64fc_16sc* pState,
     const Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_32sc(IppsFIRState64fc_32sc* pState,
     const Ipp32sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_32fc(IppsFIRState64fc_32fc* pState,
     const Ipp32fc* pDlyLine);
```

Arguments

pState	Pointer to the FIR filter state structure.
pDlyLine	Pointer to the array holding the delay line values.

Discussion

The ippsFIRGetDlyLine and ippsFIRSetDlyLine functions get and set the delay line values of a FIR filter state.

ippsFIRGetDlyLine. The function ippsFIRGetDlyLine copies the delay line values from the state structure *pState* and stores them into *pDlyLine*.

ippsFIRSetDlyLine. The function ippsFIRSetDlyLine copies the delay line values from *pDlyLine* and stores them into the state structure *pState*.

Before calling either ippsFIRGetDlyLine or ippsFIRSetDlyLine, initialize the filter state by calling the function ippsFIRInitAlloc or ippsFIRMRInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

Single-rate FIR LMS Filter Functions

The functions described in this section perform the following tasks:

- initialize a single-rate FIR LMS filter
- get and set the delay line values
- get the filter coefficients (taps) values
- perform filtering
- free dynamic memory allocated for the functions state

To use the single-rate FIR LMS adaptive filter functions, follow this general scheme:

1. Call `ippsFIRLMSInitAlloc` to initialize a single-rate FIR LMS filter.
2. Call `ippsFIRLMSOne_Direct` to make one iteration of FIR filter taps fitting with one input sample and/or call `ippsFIRLMS` to fit taps with a block of consecutive input samples.
3. Call `ippsFIRLMSGetTaps` to get the filter coefficients (taps). Call `ippsFIRLMSGetDlyLine` and `ippsFIRLMSSetDlyLine` to get and set the values in the delay line.
4. Call `ippsFIRLMSFree` to release dynamic memory associated with the FIR LMS filter.

FIRLMSInitAlloc

Initializes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.

```
IppStatus ippsFIRLMSInitAlloc_32f(IppsFIRLMSState_32f** pState,
     const Ipp32f* pTaps, int tapsLen, const Ipp32f* pDlyLine, int
     dlyLineIndex);
IppStatus ippsFIRLMSInitAlloc32f_16s(IppsFIRLMSState32f_16s** pState,
     const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine, int
     dlyLineIndex);
```

Arguments

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.

tapsLen	Number of elements in the array containing the tap values.
pDlyLine	Pointer to the array holding the delay line values. The number of elements in the array is 2**tapsLen*.
dlyLineIndex	Current index of the delay line.
pState	Address of the pointer to the state structure to be created.

Discussion

The function ippsFIRLMSInitAlloc creates and initializes a single-rate FIR LMS filter state. The function ippsFIRLMSInitAlloc copies the taps from the *tapsLen* -length array *pTaps* into the state structure *pState*. The 2**tapsLen* -length array *pDlyLine* specifies the delay line values. The current index of the delay line *pDlyLine* is defined by *dlyLineIndex*. If the pointer to the array *pDlyLine* or *pTaps* is NULL, then the corresponding values of the state structure are initialized to 0.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSFree

Closes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.

```
IppStatus ippsFIRLMSFree_32f(IppsFIRLMSState_32f* pState);
IppStatus ippsFIRLMSFree32f_16s(IppsFIRLMSState32f_16s* pState);
```

Arguments

pState	Pointer to the FIR LMS filter state structure to be closed.

Discussion

The function ippsFIRLMSFree closes the FIR LMS filter state by freeing all memory associated with a filter state created by ippsFIRLMSInitAlloc. Call ippsFIRLMSFree after filtering is completed.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSOne_Direct

Filters a single sample through a FIR LMS filter.

```
IppStatus ippsFIRLMSOne_Direct_32f(Ipp32f src, Ipp32f ref,
     Ipp32f* pDstVal, Ipp32f* pTapsInv, int tapsLen, float mu,
     Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSOne_Direct32f_16s(Ipp16s src, Ipp16s ref,
     Ipp16s* pDstVal, Ipp32f* pTapsInv, int tapsLen, float mu,
     Ipp16s* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSOne_DirectQ15_16s(Ipp16s src, Ipp16s ref,
     Ipp16s* pDstVal, Ipp32s* pTapsInv, int tapsLen, int mu15,
     Ipp16s* pDlyLine, int* pDlyLineIndex);
```

Arguments

src	Input sample to be filtered.
pDstVal	Pointer to the output sample.
ref	Reference signal sample.
pTapsInv	Pointer to the array containing the FIR filter taps to be adapted. The tap values are stored in the array in the inverse order.

tapsLen	Number of elements in the array containing the tap values.
pDlyLine	Pointer to the array holding the delay line values.
pDlyLineIndex	Pointer to the current index of the delay line.
mu	Adaptation step.
muQ15	Integer version adaptation step.

Discussion

The function ippsFIRLMSOne_Direct performs directly a single iteration of FIR filter taps adaptation. The *tapsLen* -length array *pTapsInv* contains the FIR filter taps in the inverse order. The 2**tapsLen* -length array *pDlyLine* specifies the delay line values. The *pDlyLineIndex* array specifies the current index of the delay line. The output signal is stored in *pDstVal*.

NOTE. *The adaptation error value can be computed as follows:* *err*[*n*] = *ref*[*n*] - **pDstVal*.

The function ippsFIRLMSOne_Direct performs a single iteration of FIR filter taps adaptation with the *mu* step value. The taps are floating-point numbers.

Set the taps to zero or to values close to calculated to speed up the process.

The function ippsLMSOne_Direct is to be called within cycle with the number of iterations equal to the number of input samples. You can decide what data is to be saved as a result of adaptation: either the filtered output signal or the adaptation error.

The function ippsFIRLMSOne_Direct with the Q15 suffix performs a single iteration of FIR filter taps adaptation. The taps are integer numbers. The adaptation step *muQ15* can be computed as follows:

muQ15 = (int)(*mu* *(1<<15)+0.5f)

Example 6-4 illustrates the use of the function ippsFIRLMSOne_Direct to adapt the FIR filter taps. After adaptation the taps are close to 1.0.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsSizeErr	Indicates an error when *tapsLen* is less or equal to 0.

Example 6-4 Usage of the ippsLMSOne_Direct Function

```
IppStatus firlmsone(void) {
#define LEN 200
      IppStatus st = ippStsNoErr;
      Ipp32f taps = 0, dly[2] = {0};
      Ipp32f x[LEN], y[LEN], mu = 0.05f;
      int i, indx = 0;
      /// make a const signal of amplitude 1 and noise it
      for( i=0; i<LEN; ++i ) x[i] = 0.4f * rand()/RAND_MAX + 0.8f;
      for( i=0; i<LEN-1 && ippStsNoErr==st; ++i )
      st=ippsFIRLMSOne_Direct_32f( x[i], x[i+1], y+1+i, &taps, 1, mu,
          dly, &indx );
      printf_32f("FIRLMSOne tap adapted =", &taps, 1, st );
      return st;
}
Output:
    FIRLMSOne tap adapted = 0.993872
```

FIRLMS

Filters an array through a FIR LMS filter.

```
IppStatus ippsFIRLMS_32f(const Ipp32f* pSrc, const Ipp32f* pRef,
     Ipp32f* pDst, int len, float mu, IppsFIRLMSState_32f* pState);
IppStatus ippsFIRLMS32f_16s(const Ipp16s* pSrc, const Ipp16s* pRef,
     Ipp16s* pDst, int len, float mu, IppsFIRLMSState32f_16s* pState);
```

Arguments

pState	Pointer to the FIR LMS filter state structure.
pSrc	Pointer to the input sample to be filtered.
pRef	Pointer to the reference signal
pDst	Pointer to the output signal
len	Number of elements in the array.
mu	Adaptation step.

Discussion

The function `ippsFIRLMS` filters an input array *pSrc* and performs adaptation of the taps using the reference signal *pRef* and the adaptation step *mu*.

Each of *len* iterations performed by the function consists of two main procedures. First, `ippsLMS` filters the current sample of the input signal *pSrc* and stores the result in *pDst*. Next, the function performs fitting of the current taps using the reference signal *pRef*, the computed result signal *pDst*, and the adaptation step *mu*.

Before using `ippsFIRLMS`, initialize the *pState* structure by calling `ippsFIRLMSInitAlloc`.

Example 6-5 illustrates the use of the function `ippsFIRLMS_32f` to filter a signal sample.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less or equal to 0.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

Example 6-5 Filtering with the ippsFIRLMS Function

```
IppStatus firlms(void) {
     IppStatus st;
     Ipp32f taps = 0, x[LEN], y[LEN], mu = 0.03f;
     IppsFIRLMSState_32f* ctx;
     int i;
     /// no taps and no delay line from outside
     ippsFIRLMSInitAlloc_32f( &ctx, 0, 1, 0, 0 );
     /// make a const signal of amplitude 1 and noise it
     for(i=0; i<LEN; ++i) x[i] = 0.4f * rand()/RAND_MAX + 0.8f;
     st = ippsFIRLMS_32f( x, x+1, y, LEN-1, mu, ctx);
     /// get FIR LMS tap, it must be near to 1
     ippsFIRLMSGetTaps_32f(ctx, &taps);
     ippsFIRLMSFree_32f(ctx);
     printf_32f("FIR LMS tap fitted =", &taps, 1, st);
     return st;
}
Output:
    FIR LMS tap adapted = 0.986842
```

FIRLMSGetTaps

Gets the taps of a FIR LMS filter.

```
IppStatus ippsFIRLMSGetTaps_32f(const IppsFIRLMSState_32f* pState,
     Ipp32f* pOutTaps);
IppStatus ippsFIRLMSGetTaps32f_16s(const IppsFIRLMSState32f_16s*
     pState, Ipp32f* pOutTaps);
```

Arguments

pState	Pointer to the FIR LMS filter state structure.
pOutTaps	Pointer to the array holding copies of the taps.

Discussion

The function ippsFIRLMSGetTaps copies the taps from the state structure *pState* to the *tapsLen* -length array *pOutTaps*. To set new taps in the state structure, create a new state using the function ippsFIRLMSInitAlloc.

Before calling the function ippsFIRLMSGetTaps, initialize the filter state by calling ippsFIRLMSInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSGetDlyLine, FIRLMSSetDlyLine

Gets and sets the delay line contents of a FIR LMS filter.

```
IppStatus ippsFIRLMSGetDlyLine_32f(const IppsFIRLMSState_32f* pState,
     Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSGetDlyLine32f_16s(const IppsFIRLMSState32f_16s*
     pState, Ipp16s* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSSetDlyLine_32f(IppsFIRLMSState_32f* pState,
     const Ipp32f* pDlyLine, int dlyLineIndex);
IppStatus ippsFIRLMSSetDlyLine32f_16s(IppsFIRLMSState32f_16s* pState,
     const Ipp16s* pDlyLine, int dlyLineIndex);
```

Arguments

pState	Pointer to the FIR LMS filter state structure.
pDlyLine	Pointer to the *tapsLen* -length array holding the delay line values.
pDlyLineIndex	Pointer to the array to store the current delay line index copied from *pState* by the function ippsFIRLMSGetDlyLine.
dlyLineIndex	Initial index of the delay line to be stored in *pState* by the function ippsFIRLMSSetDlyLine.

Discussion

The ippsFIRLMSGetDlyLine and ippsFIRLMSSetDlyLine functions get and set the delay line values of a FIR LMS filter state.

ippsFIRLMSGetDlyLine. The function ippsFIRLMSGetDlyLine copies the delay line values and the current delay line index from the state structure *pState*, and stores them into *pDlyLine* and *pDlyLineIndex*, respectively.

ippsFIRLMSSetDlyLine. The function ippsFIRLMSSetDlyLine copies the delay line values from *pDlyLine* and the current delay line index from *dlyLineIndex*, and stores them into the state structure *pState*.

Before calling either ippsFIRLMSGetDlyLine or ippsFIRLMSSetDlyLine, initialize the filter state by calling the function ippsFIRLMSInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

Multi-Rate FIR LMS Filter Functions

The functions described in this section perform the following tasks:

- initialize a multi-rate FIR LMS filter
- get and set the delay line values
- get and set the filter coefficients (taps) values
- set the adaptation step value
- perform filtering
- update the filter coefficients using the result of the filter operation
- free dynamic memory allocated for the functions state

To use the multi-rate FIR LMS adaptive filter functions, follow this general scheme:

1. Call ippsFIRLMSMRInitAlloc to initialize a multi-rate FIR LMS filter.
2. Call ippsFIRLMSMRPutVal a required number of times to place the input values in the delay line.
3. Call ippsFIRLMSMROne to filter the samples in the delay line.
4. Call ippsFIRLMSMRUpdateTaps to update the taps using the value of adaptation error that is based on comparison between filtered and reference signals.
5. Call ippsFIRLMSMRGetTaps and ippsFIRLMSMRSetTaps to get and set the filter coefficients (taps). Call ippsFIRLMSMRGetDlyLine and ippsFIRLMSMRSetDlyLine to get and set the values in the delay line.

6. Call `ippsFIRLMSMRFree` to release dynamic memory associated with the FIR LMS filter.

FIRLMSMRInitAlloc

Initializes an adaptive multi-rate FIR filter that uses the least mean squares (LMS) algorithm.

```
IppStatus ippsFIRLMSMRInitAlloc32s_16s(IppsFIRLMSMRState32s_16s**
     pState, const Ipp32s* pTaps, int tapsLen, const Ipp16s* pDlyLine,
     int dlyLineIndex, int dlyStep, int updateDly, int mu);
IppStatus ippsFIRLMSMRInitAlloc32sc_16sc(IppsFIRLMSMRState32sc_16sc**
     pState, const Ipp32sc* pTaps, int tapsLen, const Ipp16sc* pDlyLine,
     int dlyLineIndex, int dlyStep, int updateDly, int mu);
```

Arguments

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
pDlyLine	Pointer to the array holding the delay line values. The number of elements in the array is *tapsLen*dlyStep + updateDly*.
dlyLineIndex	Current index of the delay line.
dlyStep	Multi-rate down factor applied to delay line values.
updateDly	Value of adaptation delay in samples.
mu	Adaptation step.
pState	Address of the pointer to the filter state structure to be created.

Discussion

The function ippsFIRLMSMRInitAlloc creates and initializes a multi-rate FIR LMS filter state. The function copies the filter coefficients from *tapsLen*-length array *pTaps* into the state structure *pState*. The array *pDlyLine* specifies the delay line values. The structure is initialized by the downsampling factor over the delay line *dlyStep*, by the adaptation delay value *updateDly*, and by the adaptation step value *mu*. The function returns the pointer in the output parameter *pState* and also the operation status value.

The function uses copies of the tap values during the filtering procedure. The initial values passed to the function may be obtained from the previous filtering procedure.If the pointer *pTaps* is NULL, then the filter coefficient values are set to zero.

Copies of the input samples are used in the filtering procedure. Values in the delay line represent data in the same format as the input data to be filtered. If the pointer *pDlyLine* is NULL, then the filter delay line values are set to zero.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSMRFree

Closes an adaptive multi-rate FIR filter that uses the least mean squares algorithm.

```
IppStatus ippsFIRLMSMRFree32s_16s(IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMRFree32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState);
```

Arguments

pState	Pointer to the multi-rate FIR LMS filter state structure to be closed.

Discussion

The function ippsFIRLMSMRFree closes the multi-rate FIR LMS filter state by freeing all memory associated with a filter state created by ippsFIRLMSMRInitAlloc. Call ippsFIRLMSMRFree after filtering is completed.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSMRPutVal

Places the input value in the delay line.

```
IppStatus ippsFIRLMSMRPutVal32s_16s(Ipp16s val,
     IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMRPutVal32sc_16sc(Ipp16sc val,
     IppsFIRLMSMRState32sc_16sc* pState);
```

Arguments

pState	Pointer to the filter state structure.
val	Value of the input sample.

Discussion

The function ippsFIRLMSMRPutVal places the value of the input sample *val* into the delay line, thus preparing the filter with the state structure *pState* for the filtering procedure.

Before calling the ippsFIRLMSMRPutVal, initialize the filter state by calling the function ippsFIRLMSMRInitAlloc.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSMROne

Filters data placed in the delay line.

```
IppStatus ippsFIRLMSMROne32s_16s(Ipp32s* pDstVal,
     IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMROne32sc_16sc(Ipp32sc* pDstVal,
     IppsFIRLMSMRState32sc_16sc* pState);
```

Arguments

pState	Pointer to the filter state structure.
pDstVal	Pointer to the output signal value.

Discussion

The function `FIRLSMMROne` filters the samples placed in the delay line using the filter coefficients stored in the filter state structure *pState*. The resulting value is placed in the *pDstVal*. The downsampling factor *dlyStep* defines the number of samples that are filtered. The filter coefficients are not updated.

The filtering procedure can be described as a FIR filter operation (here the input sample to be filtered is denoted *x(n)*, the taps are denoted *h(i)*, and the return value is *y(n)*):

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-(i \cdot dlyStep))$$

Note that the function operates with values stored in the delay line that are copies of the input samples.
Before calling the function ippsFIRLMSMROne, initialize the filter state by calling ippsFIRLMSMRInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState* or *pDstVal* pointers are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSMROneVal

Filters one input value.

```
IppStatus ippsFIRLMSMROneVal32s_16s(Ipp16s val, Ipp32s* pDstVal,
     IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMROneVal32sc_16sc(Ipp16sc val, Ipp32sc* pDstVal,
     IppsFIRLMSMRState32sc_16sc* pState);
```

Arguments

pState	Pointer to the filter state structure.
pDstVal	Pointer to the output signal value.
val	Value of the input signal sample.

Discussion

The function ippsFIRLMSMROneVal places one input sample *val* into the delay line and filters it using the filter coefficients specified in the filter state structure *pState*. The result value is stored in the *pDstVal*.

Before calling the function ippsFIRLMSMROneVal, initialize the filter state by calling ippsFIRLMSMRInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState* or *pDstVal* pointers are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSMRUpdateTaps

Updates the filter coefficients using the adaptation error value.

```
IppStatus ippsFIRLMSMRUpdateTaps32s_16s(Ipp32s errVal,
     IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMRUpdateTaps32sc_16sc(Ipp32sc errVal,
     IppsFIRLMSMRState32sc_16sc* pState);
```

Arguments

pState	Pointer to the filter state structure.
errVal	Adaptation error value.l

Discussion

The ippsFIRLMSMRUpdateTaps function updates the filter coefficients that are stored in the filter state structure *pState*. It is assumed that the filter operation was performed and the adaptation error value *errVal* was computed before calling this function. The adaptation error value is computed outside the function as the difference between output and reference signals. Updated filter coefficients are defined as

$$h_{n+1}(i) = h_n(i) + mu \cdot errVal \cdot x(n - (i \cdot dlyStep) - updateDly)$$

where $h_{n+1}(i)$ denotes new taps, $h_n(i)$ denotes initial taps, and *mu*, *errVal* and *updateDly* are the adaptation step, adaptation error value and adaptation delay, respectively.
Before calling the function ippsFIRLMSMRUpdateTaps, initialize the filter state by calling the function ippsFIRLMSMRInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSMRGetTaps, FIRLMSMRSetTaps

Gets and sets taps of a multi-rate FIR LMS filter.

```
IppStatus ippsFIRLMSMRGetTaps32s_16s(
     IppsFIRLMSMRState32s_16s* pState, Ipp32s* pOutTaps);
IppStatus ippsFIRLMSMRGetTaps32sc_16sc(
     IppsFIRLMSMRState32sc_16sc* pState, Ipp32sc* pOutTaps);
IppStatus ippsFIRLMSMRSetTaps32s_16s(
     IppsFIRLMSMRState32s_16s* pState, const Ipp32s* pInTaps);
IppStatus ippsFIRLMSMRSetTaps32sc_16sc(
     IppsFIRLMSMRState32sc_16sc* pState, const Ipp32sc* pInTaps);
```

Arguments

pState	Pointer to the filter state structure.
pOutTaps	Pointer to the array holding copies of the taps.
pInTaps	Pointer to the array holding new tap values.

Discussion

The ippsFIRLMSMRGetTaps and ippsFIRLMSMRSetTaps functions get and set the filter coefficients.

ippsFIRLMSMRGetTaps. The ippsFIRLMSMRGetTaps function copies the values of the filter coefficients stored in the filter state structure *pState* to the array pointed by the *pOutTaps* pointer.

ippsFIRLMSMRSetTaps. The ippsFIRLMSMRSetTaps function sets the the filter coefficients stored in the filter state structure *pState* to the new values stored in an array pointed by the *pInTaps* pointer. If the pointer is NULL, then the filter coefficients values are set to zero.

Before calling either ippsFIRLMSMRGetTaps or ippsFIRLMSMRSetTaps, initialize the filter state by calling the function ippsFIRLMSMRInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState* or *pOutTaps* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSMRGetTapsPointer

Returns the pointer to the filter coefficients.

```
IppStatus ippsFIRLMSMRGetTapsPointer32s_16s(
     IppsFIRLMSMRState32s_16s* pState, Ipp32s** pTaps);
IppStatus ippsFIRLMSMRGetTapsPointer32sc_16sc(
     IppsFIRLMSMRState32sc_16sc* pState, Ipp32sc** pTaps);
```

Arguments

pState	Pointer to the filter state structure.
pTaps	Pointer to the variable that contains the pointer to the tap values.

Discussion

The ippsFIRLMSMRGetTapsPointer function writes the pointer to the filter coefficients stored in the filter state structure *pState* to the variable pointed by *pTaps*.

CAUTION. To get the pointer to tap values directly is faster than to copy them using the ippsFIRLMSMRGetTaps function, but this operation may be error-prone.

Before calling the function ippsFIRLMSMRGetTapsPointer, initialize the filter state by calling the function ippsFIRLMSMRInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState* or *pTaps* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSMRGetDlyLine, FIRLMSMRSetDlyLine

Gets and sets the delay line contents of a multi-rate FIR LMS filter state.

```
IppStatus ippsFIRLMSMRGetDlyLine32s_16s(IppsFIRLMSMRState32s_16s*
      pState, Ipp16s* pOutDlyLine, int* pOutDlyLineIndex);
IppStatus ippsFIRLMSMRGetDlyLine32sc_16sc(IppsFIRLMSMRState32sc_16sc*
      pState, Ipp16sc* pOutDlyLine, int* pOutDlyLineIndex);
IppStatus ippsFIRLMSMRSetDlyLine32s_16s(IppsFIRLMSMRState32s_16s*
      pState, const Ipp16s* pInDlyLine, int dlyLineIndex);
IppStatus ippsFIRLMSMRSetDlyLine32sc_16sc(IppsFIRLMSMRState32sc_16sc*
      pState, const Ipp16sc* pInDlyLine, int dlyLineIndex);
```

Arguments

pState	Pointer to the filter state structure.
pOutDlyLine	Pointer to the array containing the copies of delay line values. The number of elements in the array is *tapsLen*dlyStep+updateDly*.
pInDlyLine	Pointer to the array containing the new delay line values.The number of elements in the array is *tapsLen*dlyStep+updateDly*.
pOutDlyLineIndex	Pointer to the array to store the current delay line index copied from *pState*.
dlyLineIndex	Initial index of the delay line to be stored in *pState* .

Discussion

The ippsFIRLMSMRGetDlyLine and ippsFIRLMSMSetDlyLine functions get and set the delay line values of a multi-rate FIR LMS filter state.

ippsFIRLMSMRGetDlyLine. The ippsFIRLMSMRGetDlyLine function copies the delay line values and the current delay line index from the state structure *pState*, and stores them into *pOutDlyLine* and *pOutDlyLineIndex*, respectively.

ippsFIRLMSMRSetDlyLine. The ippsFIRLMSMRSetDlyLine function copies the new values stored in the *pInDlyLine* and corresponding delay line index from *dlyLineIndex* and stores them into the state structure *pState*. If the pointer *pInDlyLine* is NULL, the delay line values are set to zero.

Before calling either ippsFIRLMSMRGetDlyLine or ippsFIRLMSMRSetDlyLine, initialize the filter state by calling the function ippsFIRLMSMRInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the *pState*, *pOutDlyLine*, *pOutDlyLineIndex* pointers is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSMRGetDlyVal

Gets one delay line values from the specified position.

```
IppStatus ippsFIRLMSMRGetDlyVal32s_16s(IppsFIRLMSMRState32s_16s*
     pState, Ipp16s* pOutVal, int index);
IppStatus ippsFIRLMSMRGetDlyVal32sc_16sc(IppsFIRLMSMRState32sc_16sc*
     pState, Ipp16sc* pOutVal, int index);
```

Arguments

pState	Pointer to the filter state structure.
pOutVal	Pointer to the copied delay line value.
index	Index of the required delay line value.

Discussion

The ippsFIRLMSMRGetDlyVal function copies from the filter state structure *pState* one value to the *pOurVal*. The position of this sample in the delay line is specified by *index* (which means that *index* iterations ago the sample was placed into the delay line).

Before calling the function ippsFIRLMSMRGetDlyVal, initialize the filter state by calling the function ippsFIRLMSMRInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState* or *pOutVal* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSMRSetMu

Sets the adaptation step.

```
IppStatus ippsFIRLMSMRSetMu32s_16s(IppsFIRLMSMRState32s_16s* pState,
     const int mu);
IppStatus ippsFIRLMSMRSetMu32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
     const int mu);
```

Arguments

mu	New adaptation step
pState	Pointer to the filter state structure.

Discussion

The `ippsFIRLMSMRSetMu` function updates the adaptation step stored in *pState* with the new value *mu*.

Before calling the function `ippsFIRLMSMRSetMu`, initialize the filter state by calling the function `ippsFIRLMSMRInitAlloc`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

IIR Filter Functions

The functions described in this section initialize an infinite impulse response (IIR) filter and perform filtering.

To initialize and use an IIR filter, follow this general scheme:

1. Call `ippsIIRInitAlloc` to initialize the filter as an arbitrary order IIR filter or call `ippsIIRInitAlloc_BiQuad` to initialize the filter as a cascade of biquads.
2. Call `ippsIIROne` repeatedly to filter a single sample through an IIR filter or call `ippsIIR` to filter consecutive samples one at a time.
3. Call `ippsIIRGetDlyLine` and `ippsIIRSetDlyLine` to get and set the delay line values in the IIR state structure.
4. After all filtering is complete, call `ippsIIRFree` to release dynamic memory associated with the filter.

IIRInitAlloc, IIRInitAlloc_BiQuad

Initializes an infinite impulse response filter.

```
IppStatus ippsIIRInitAlloc_32f(IppsIIRState_32f** pState,
     const Ipp32f* pTaps, int order, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc_32fc(IppsIIRState_32fc** pState,
     const Ipp32fc* pTaps, int order, const Ipp32fc* pDlyLine);
IppStatus ippsIIRInitAlloc32f_16s(IppsIIRState32f_16s** pState,
     const Ipp32f* pTaps, int order, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc32fc_16sc(IppsIIRState32fc_16sc** pState,
     const Ipp32fc* pTaps, int order, const Ipp32fc* pDlyLine);

IppStatus ippsIIRInitAlloc_64f(IppsIIRState_64f** pState,
     const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc_64fc(IppsIIRState_64fc** pState,
     const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc64f_32f(IppsIIRState64f_32f** pState,
     const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_32fc(IppsIIRState64fc_32fc** pState,
     const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc64f_32s(IppsIIRState64f_32s** pState,
     const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc64fc_32sc(IppsIIRState64fc_32sc** pState,
     const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);

IppStatus ippsIIRInitAlloc64f_16s(IppsIIRState64f_16s** pState,
     const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc64fc_16sc(IppsIIRState64fc_16sc** pState,
     const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);


IppStatus ippsIIRInitAlloc32sc_16sc(IppsIIRState32sc_16sc** pState,
     const Ipp32sc* pTaps, int order, int tapsFactor,
     const Ipp32sc* pDlyLine);

IppStatus ippsIIRInitAlloc32sc_16sc32fc(IppsIIRState32sc_16sc** pState,
     const Ipp32fc* pTaps, int order, const Ipp32sc* pDlyLine);


IppStatus ippsIIRInitAlloc_BiQuad_32f(IppsIIRState_32f** pState,
     const Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine);

IppStatus ippsIIRInitAlloc_BiQuad_32fc(IppsIIRState_32fc** pState,
     const Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine);

IppStatus ippsIIRInitAlloc32f_BiQuad_16s(IppsIIRState32f_16s** pState,
     const Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine);

IppStatus ippsIIRInitAlloc32fc_BiQuad_16sc(IppsIIRState32fc_16sc**
     pState, const Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine);


IppStatus ippsIIRInitAlloc_BiQuad_64f(IppsIIRState_64f** pState,
     const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc_BiQuad_64fc(IppsIIRState_64fc** pState,
     const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);

IppStatus ippsIIRInitAlloc64f_BiQuad_32f(IppsIIRState64f_32f** pState,
     const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc64fc_BiQuad_32fc(IppsIIRState64fc_32fc**
     pState, const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);

IppStatus ippsIIRInitAlloc64f_BiQuad_32s(IppsIIRState64f_32s** pState,
     const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc64fc_BiQuad_32sc(IppsIIRState64fc_32sc**
     pState, const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);

IppStatus ippsIIRInitAlloc64f_BiQuad_16s(IppsIIRState64f_16s** pState,
     const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc64fc_BiQuad_16sc(IppsIIRState64fc_16sc**
     pState, const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc32s_16s(IppsIIRState32s_16s** pState,
     const Ipp32s* pTaps, int order, int tapsFactor, const Ipp32s*
     pDlyLine);
IppStatus ippsIIRInitAlloc32s_16s32f(IppsIIRState32s_16s** pState,
     const Ipp32f* pTaps, int order, const Ipp32s* pDlyLine);

IppStatus ippsIIRInitAlloc32s_BiQuad_16s(IppsIIRState32s_16s** pState,
     const Ipp32s* pTaps, int numBq, int tapsFactor, const Ipp32s* pDlyLine);
IppStatus ippsIIRInitAlloc32s_BiQuad_16s32f(IppsIIRState32s_16s**
     pState, const Ipp32f* pTaps, int numBq, const Ipp32s* pDlyLine);

IppStatus ippsIIRInitAlloc32sc_BiQuad_16sc(IppsIIRState32sc_16sc**
     pState, const Ipp32sc* pTaps, int numBq, int tapsFactor,
     const Ipp32sc* pDlyLine);
IppStatus ippsIIRInitAlloc32sc_BiQuad_16sc32fc(IppsIIRState32sc_16sc**
     pState, const Ipp32fc* pTaps, int numBq, const Ipp32sc* pDlyLine);
```

Arguments

pTaps	Pointer to the array containing the taps. The number of elements in the array is 2*(*order*+1) for arbitrary filters and 6**numBq* for BQ filters.
tapsFactor	Scale factor for the taps of Ipp32s data type (for integer versions only)
numBq	Number of cascades of biquads. The *numBq* argument is used by the function ippsIIRBQInit.
order	Order of the IIR filter. This argument is used by the function ippsIIRInit.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order* for arbitrary filters and 2**numBq* for BQ filters.
pState	Pointer to the IIR state structure to be created.

Discussion

The functions ippsIIRInitAlloc and ippsIIRInitAlloc_BiQuad create and initialize an arbitrary or biquad (BQ) IIR filter state, respectively. The initialization functions copy the taps from the array *pTaps* into the state structure *pState*. To scale integer taps use the *tapsFactor* value. The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not NULL, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

If the state is not created, the initialization function returns an error status.

ippsIIRInitAlloc. The function ippsIIRInitAlloc initializes the taps and the delay line in the state structure of an arbitrary order IIR filter. The *order* -length array *pDlyLine* specifies the delay line values. The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The 2*(*order* + 1) -length array *pTaps* specifies the taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{order}, A_0, A_1, \ldots, A_{order}$

$A_0 \neq 0$

Before using ippsIIRInitAlloc or ippsIIRInitAlloc_BiQuad with the 32s suffixes called with integer taps, convert the taps from floating-point into integer data type by calling ippsConvert.

The initialization functions with the 32s_32f suffixes called with floating-point taps convert the taps into integer data type automatically.

In both cases the data is converted into integer type with scaling to provide better precision. Example 6-7 shows how to convert floating-point taps into integer data type.

ippsIIRInitAlloc_BiQuad. The function ippsIIRInitAlloc_BiQuad initializes the taps and the delay line in the state structure of a biquad IIR filter; that is, defined by a cascade of biquads. The 2*numBq -length array *pDlyLine* specifies the delay line values. The number of cascades of biquads is defined by the *numBq* value. The 6*numBq -length array *pTaps* specifies the taps arranged in the array as follows:

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; \quad B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \ldots A_{numBq\text{-}1,2}$

$A_{n,0} \neq 0, B_{n,0} \neq 0$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsMemAllocErr`	Indicates an error when no memory allocated.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are `NULL`.
`ippStsIIROrderErr`	Indicates an error when *`order`* is negative or *`numBq`* is less or equal to 0.
`ippStsDivByZeroErr`	Indicates an error when A_0, $A_{n,0}$ or $B_{n,0}$ is equal to 0.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

IIRFree

Closes an IIR filter state.

```
IppStatus ippsIIRFree_32f(IppsIIRState_32f* pState);
IppStatus ippsIIRFree_64f(IppsIIRState_64f* pState);
IppStatus ippsIIRFree32s_16s(IppsIIRState32s_16s* pState);
IppStatus ippsIIRFree32f_16s(IppsIIRState32f_16s* pState);
IppStatus ippsIIRFree64f_16s(IppsIIRState64f_16s* pState);
IppStatus ippsIIRFree64f_32s(IppsIIRState64f_32s* pState);
IppStatus ippsIIRFree64f_32f(IppsIIRState64f_32f* pState);
IppStatus ippsIIRFree_32fc(IppsIIRState_32fc* pState);
IppStatus ippsIIRFree_64fc(IppsIIRState_64fc* pState);
IppStatus ippsIIRFree32sc_16sc(IppsIIRState32sc_16sc* pState);
IppStatus ippsIIRFree32fc_16sc(IppsIIRState32fc_16sc* pState);
IppStatus ippsIIRFree64fc_16sc(IppsIIRState64fc_16sc* pState);
IppStatus ippsIIRFree64fc_32sc(IppsIIRState64fc_32sc* pState);
IppStatus ippsIIRFree64fc_32fc(IppsIIRState64fc_32fc* pState);
```

Arguments

`pState`	Pointer to an IIR filter state structure to be closed.

Discussion

The function ippsIIRFree closes the IIR filter state by freeing all memory associated with a filter state created by ippsIIRInitAlloc or ippsIIRBQInitAlloc. Call ippsIIRFree after filtering is completed.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

IIROne

Filters a single sample through an IIR filter.

```
IppStatus ippsIIROne_32f(Ipp32f src, Ipp32f* pDstVal,
     IppsIIRState_32f* pState);
IppStatus ippsIIROne_64f(Ipp64f src, Ipp64f* pDstVal,
     IppsIIRState_64f* pState);
IppStatus ippsIIROne64f_32f(Ipp32f src, Ipp32f* pDstVal,
     IppsIIRState64f_32f* pState);
IppStatus ippsIIROne_32fc(Ipp32fc src, Ipp32fc* pDstVal,
     IppsIIRState_32fc* pState);
IppStatus ippsIIROne_64fc(Ipp64fc src, Ipp64fc* pDstVal,
     IppsIIRState_64fc* pState);
IppStatus ippsIIROne64fc_32fc(Ipp32fc src, Ipp32fc* pDstVal,
     IppsIIRState64fc_32fc* pState);

IppStatus ippsIIROne32s_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
     IppsIIRState32s_16s* pState, int scaleFactor);
IppStatus ippsIIROne32f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
     IppsIIRState32f_16s* pState, int scaleFactor);
IppStatus ippsIIROne64f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
     IppsIIRState64f_16s* pState, int scaleFactor);
```

```
IppStatus ippsIIROne64f_32s_Sfs(Ipp32s src, Ipp32s* pDstVal,
     IppsIIRState64f_32s* pState, int scaleFactor);


IppStatus ippsIIROne32sc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
     IppsIIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne32fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
     IppsIIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne64fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
     IppsIIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne64fc_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
     IppsIIRState64fc_32sc* pState, int scaleFactor);
```

Arguments

pState	Pointer to the IIR filter state structure.
src	Input sample to be filtered by the function ippsIIROne.
pDstVal	Pointer to the output sample filtered by the function ippsIIROne.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsIIROne filters a single sample *src* through an IIR filter with real taps, and stores the result in *pDstVal*. The filter parameters are specified in *pState*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Do not modify the *scaleFactor* value unless the state structure is changed.

Before calling either the ippsIIR or ippsFIROne function, initialize the filter state by calling ippsIIRInitAlloc or ippsIIRBQInitAlloc. Specify the number of taps *tapsLen*, the tap values in *pTaps*, the delay line values in *pDlyLine*, and the *order* or *numBq* value beforehand.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.

`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect.

IIR

Filters a block of samples through an IIR filter.

```
IppStatus ippsIIR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
      IppsIIRState_32f* pState);
IppStatus ippsIIR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
      IppsIIRState_64f* pState);
IppStatus ippsIIR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
      IppsIIRState_32fc* pState);
IppStatus ippsIIR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
      IppsIIRState_64fc* pState);
IppStatus ippsIIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
      IppsIIRState64f_32f* pState);
IppStatus ippsIIR64fc_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
       IppsIIRState64fc_32fc* pState);

IppStatus ippsIIR_32f_I(Ipp32f* pSrcDst, int len,
      IppsIIRState_32f* pState);
IppStatus ippsIIR_64f_I(Ipp64f* pSrcDst, int len,
      IppsIIRState_64f* pState);
IppStatus ippsIIR_32fc_I(Ipp32fc* pSrcDst, int len,
      IppsIIRState_32fc* pState);
IppStatus ippsIIR_64fc_I(Ipp64fc* pSrcDst, int len,
      IppsIIRState_64fc* pState);
IppStatus ippsIIR64f_32f_I(Ipp32f* pSrcDst, int len,
      IppsIIRState64f_32f* pState);
IppStatus ippsIIR64fc_32fc_I(Ipp32fc* pSrcDst, int len,
      IppsIIRState64fc_32fc* pState);

IppStatus ippsIIR32s_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
      IppsIIRState32s_16s* pState, int scaleFactor);
```

```
IppStatus ippsIIR32f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
     IppsIIRState32f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
     IppsIIRState64f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len,
     IppsIIRState64f_32s* pState, int scaleFactor);

IppStatus ippsIIR32sc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, IppsIIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsIIR32fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, IppsIIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, IppsIIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst,
     int len, IppsIIRState64fc_32sc* pState, int scaleFactor);
IppStatus ippsIIR32s_16s_ISfs(Ipp16s* pSrcDst, int len,
     IppsIIRState32s_16s* pState, int scaleFactor);
IppStatus ippsIIR32f_16s_ISfs(Ipp16s* pSrcDst, int len,
     IppsIIRState32f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_16s_ISfs(Ipp16s* pSrcDst, int len,
     IppsIIRState64f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_32s_ISfs(Ipp32s* pSrcDst, int len,
     IppsIIRState64f_32s* pState, int scaleFactor);
IppStatus ippsIIR32fc_16sc_ISfs(Ipp16sc* pSrcDst, int len,
     IppsIIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR32sc_16sc_ISfs(Ipp16sc* pSrcDst, int len,
     IppsIIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_16sc_ISfs(Ipp16sc* pSrcDst, int len,
     IppsIIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_32sc_ISfs(Ipp32sc* pSrcDst, int len,
     IppsIIRState64fc_32sc* pState, int scaleFactor);
```

Arguments

pState	Pointer to the IIR filter state structure.

pSrc	Pointer to the input array to be filtered by the function ippsIIR.
pDst	Pointer to the output array filtered by the function ippsIIR.
pSrcDst	Pointer to the input and output array (for the in-place operation) to be filtered by the function ippsIIR.
len	Number of samples to be filtered by the function ippsIIR.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsIIR filters *len* samples in the input array *pSrc* or *pSrcDst* through an IIR filter with real taps, and stores the results in *pDst* or *pSrcDst*, respectively. The filter parameters are specified in *pState*.The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Do not modify the *scaleFactor* value unless the state structure is changed.

Before calling ippsIIR, initialize the filter state by calling ippsIIRInitAlloc or ippsIIRBQInitAlloc. Specify the number of taps *tapsLen*, the tap values in *pTaps*, the delay line values in *pDlyLine*, and the *order* or *numBq* value beforehand.

Example 6-6 illustrates using ippsIIR_32f to suppress a 60 Hz signal.

Example 6-7 illustrates using ippsIIR to filter a sample. The function ippsCnvrt_64f32s_Sfs converts floating-point taps into integer data type before calling ippsIIRInitAlloc_32s.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsSizeErr	Indicates an error when *len* is less or equal to 0.

`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect

Example 6-6 Using the ippsIIR_32f Function to Suppress a 60 Hz Signal

```
IppStatus iir( void ) {
#undef NUMITERS
#define NUMITERS 150
     int n;
     IppStatus status;
     IppsIIRState_32f *ctx;
     Ipp32f *x = ippsMalloc_32f( NUMITERS ), *y = ippsMalloc_32f( NUMITERS );
     /// A second-order notch filter having notch freq at 60 Hz
     const float taps[] = {
        0.940809f,-1.105987f,0.940809f,1,-1.105987f,0.881618f
     };
     /// generate a signal having 60 Hz freq sampled with 400 Hz freq
     for(n=0;n<NUMITERS;++n)x[n]=(float)sin(IPP_2PI *n *60 /400);
     ippsIIRInitAlloc_32f( &ctx, taps, 2, NULL );
     status = ippsIIR_32f( x, y, NUMITERS, ctx );
     printf_32f( " IIR 32f output+120 =", y+120, 5, status );
     ippsIIRFree_32f( ctx );
     ippsFree( y );
     ippsFree( x );
     return status;
}
Output:
    IIR 32f output + 120 =  -0.000094 0.000339 0.000458 0.000208 -0.000173
Matlab* Analog:
    >> B = [0.940809,-1.105987,0.940809]; A = [1,-1.105987,0.881618];
    n = 0:150; x = sin(2*pi*n*60/400); y = filter(B,A,x); y(121:125)
```

Example 6-7 Using the ippsIIR Function to Filter a Sample

```
IppStatus iir16s( void ) {
#undef NUMITERS
#define NUMITERS 150
     int n, tapsfactor = 30;
     IppStatus status;
     IppsIIRState32s_16s *ctx;
     Ipp16s *x = ippsMalloc_16s( NUMITERS ), *y = ippsMalloc_16s( NUMITERS );
     /// A second-order notch filter having notch freq at 60 Hz
     Ipp64f taps[6] = {
        0.940809f,-1.105987f,0.940809f,1,-1.105987f,0.881618f
     };
     Ipp32s taps32s[6];
     Ipp64f tmax, tmp[6];
     ippsAbs_64f( taps, tmp, 6 );
     ippsMax_64f( tmp, 6, &tmax );
     tapsfactor = 0;
     if( tmax > IPP_MAX_32S )
        while( (tmax/=2) > IPP_MAX_32S ) ++tapsfactor;
     else
        while( (tmax*=2) < IPP_MAX_32S ) --tapsfactor;
     if(tapsfactor >0 )
        ippsDivC_64f_I( (float)(1<<(++tapsfactor)), taps, 6 );
     else if( tapsfactor < 0 )
        ippsMulC_64f_I( (float)(1<<(-(tapsfactor))), taps, 6 );
     ippsConvert_64f32s_Sfs( taps, taps32s, 6, ippRndNear, 0 );
     /// generate a signal of 60 Hz freq that is sampled with 400 Hz freq
     for(n=0; n<NUMITERS; ++n) x[n] = (Ipp16s)(1000*sin(IPP_2PI*n*60/400));
     ippsIIRInitAlloc32s_16s( &ctx, taps32s, 2, tapsfactor, NULL );
     status = ippsIIR32s_16s_Sfs( x, y, NUMITERS, 0, ctx );
     printf_16s( " IIR 32s output+120 =", y+120, 5, status );
     ippsIIRFree32s_16s( ctx );
     ippsFree( y );
     ippsFree( x );
     return status;
}
Output:
    IIR 32s output + 120 =  0 0 0 0 0
```

IIRGetDlyLine, IIRSetDlyLine

Gets and sets the delay line contents of an IIR filter state.

```
IppStatus ippsIIRGetDlyLine_32f(const IppsIIRState_32f* pState,
      Ipp32f* pDlyLine);
IppStatus ippsIIRGetDlyLine_64f(const IppsIIRState_64f* pState,
      Ipp64f* pDlyLine);
IppStatus ippsIIRGetDlyLine_32fc(const IppsIIRState_32fc* pState,
      Ipp32fc* pDlyLine);
IppStatus ippsIIRGetDlyLine_64fc(const IppsIIRState_64fc* pState,
      Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine32f_16s(const IppsIIRState32f_16s* pState,
      Ipp32f* pDlyLine);
IppStatus ippsIIRGetDlyLine64f_16s(const IppsIIRState64f_16s* pState,
      Ipp64f* pDlyLine);
IppStatus ippsIIRGetDlyLine64f_32s(const IppsIIRState64f_32s* pState,
      Ipp64f* pDlyLine);
IppStatus ippsIIRGetDlyLine64f_32f(const IppsIIRState64f_32f* pState,
      Ipp64f* pDlyLine);
IppStatus ippsIIRGetDlyLine32s_16s(const IppsIIRState32s_16s* pState,
      Ipp32s* pDlyLine);
IppStatus ippsIIRGetDlyLine32sc_16sc(const IppsIIRState32sc_16sc*
      pState, Ipp32sc* pDlyLine);
IppStatus ippsIIRGetDlyLine32fc_16sc(const IppsIIRState32fc_16sc*
      pState, Ipp32fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_16sc(const IppsIIRState64fc_16sc*
      pState, Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_32sc(const IppsIIRState64fc_32sc*
      pState, Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_32fc(const IppsIIRState64fc_32fc*
      pState, Ipp64fc* pDlyLine);
```

```
IppStatus ippsIIRSetDlyLine_32f(IppsIIRState_32f* pState,
     const Ipp32f* pDlyLine);
IppStatus ippsIIRSetDlyLine_64f(IppsIIRState_64f* pState,
     const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine_32fc(IppsIIRState_32fc* pState,
     const Ipp32fc* pDlyLine);
IppStatus ippsIIRSetDlyLine_64fc(IppsIIRState_64fc* pState,
     const Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine32s_16s(IppsIIRState32s_16s* pState,
     const Ipp32s* pDlyLine);
IppStatus ippsIIRSetDlyLine32f_16s(IppsIIRState32f_16s* pState,
     const Ipp32f* pDlyLine);
IppStatus ippsIIRSetDlyLine64f_16s(IppsIIRState64f_16s* pState,
     const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine64f_32s(IppsIIRState64f_32s* pState,
     const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine64f_32f(IppsIIRState64f_32f* pState,
     const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine32sc_16sc(IppsIIRState32sc_16sc* pState,
     const Ipp32sc* pDlyLine);
IppStatus ippsIIRSetDlyLine32fc_16sc(IppsIIRState32fc_16sc* pState,
     const Ipp32fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_16sc(IppsIIRState64fc_16sc* pState,
     const Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_32sc(IppsIIRState64fc_32sc* pState,
     const Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_32fc(IppsIIRState64fc_32fc* pState,
     const Ipp64fc* pDlyLine);
```

Arguments

pState	Pointer to the IIR filter state structure.
pDlyLine	Pointer to the array holding the delay line values. The number of elements in the array is *order* for arbitrary filters and 2**numBq* for BQ filters. If the pointer is NULL, then the delay line values in the state structure are initialized to zero.

Discussion

The functions ippsIIRGetDlyLine and ippsIIRSetDlyLine get and set the delay line values of a IIR filter state.

ippsIIRGetDlyLine. The function ippsIIRGetDlyLine copies the delay line values from the state structure *pState* and stores them into the arrays *pDlyLine*.

ippsIIRSetDlyLine. The function ippsIIRSetDlyLine copies the delay line values from *pDlyLine* and stores them into the state structure *pState*.

Before calling either ippsIIRGetDlyLine or ippsIIRSetDlyLine, initialize the filter state by calling the function ippsIIRInitAlloc or ippsIIPBQInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

Median Filter Functions

Median filters are nonlinear rank-order filters based on replacing each element of the input array with the median value, taken over the fixed neighborhood (mask) of the processed element. These filters are extensively used in image and signal processing applications. Median filtering removes impulsive noise, while keeping the signal blurring to the minimum. Typically mask size (or window width) is set to odd value which ensures simple function implementation and low output signal bias. In the IPP median function implementation, the mask is always centered at the input element for which the median value is computed. You can use an even mask size in function calls as well, but internally it will be changed to odd by subtracting 1. Another specific feature of the median function implementation in the IPP is that elements outside the input array, which are needed to determine the median value for “border” elements, are set to be equal to the corresponding edge element of the input array, i.e. are padded by cloning the edge element.

FilterMedian

Computes median values for each input array element.

```
IppStatus ippsFilterMedian_8u(const Ipp8u* pSrc, Ipp8u* pDst,
     int len, int maskSize);
IppStatus ippsFilterMedian_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, int maskSize);
IppStatus ippsFilterMedian_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len, int maskSize);
IppStatus ippsFilterMedian_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len, int maskSize);

IppStatus ippsFilterMedian_8u_I(Ipp8u* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_16s_I(Ipp16s* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_32f_I(Ipp32f* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_64f_I(Ipp64f* pSrcDst, int len, int maskSize);
```

Arguments

pSrcDst	Pointer to the input and output array (for the in-place operation).
pSrc	Pointer to the input array to be filtered by the function ippsFilterMedian.
pDst	Pointer to the output array filtered by the function ippsFilterMedian.
len	Number of elements in the array.
maskSize	Median mask size, must be a positive integer. If an even value is specified, the function subtracts 1 and uses the odd value of the filter mask for median filtering.

Discussion

The function ippsFilterMedian computes median values for each element of the input array *pSrc* or *pSrcDst*, and stores the result in *pDst* or *pSrcDst*, respectively.

NOTE. *The value of a non-existent point is equal to the last point value, for example: x*[-1]=*x*[0] *or x*[*len*]=*x*[*len*-1].

Example 6-8 illustrates using ippsFilterMedian_16s_I for single-rate filtering.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsSizeErr	Indicates an error when *len* is less or equal to 0.
ippStsEvenMedianMaskSize	Indicates a warning when the median mask length is even.

Example 6-8 Single-Rate Filtering with the ippsFilterMedian Function

```
void median(void) {
      Ipp16s x[8] = {1,2,127,4,5,0,7,8};
      IppStatus status = ippsFilterMedian_16s_I(x, 8, 3);
      printf_16s("median =", x,8, status);
}
Output:
    median =  1 2 4 5 4 5 7 8
Matlab* Analog:
    >> x = [1 2 127 4 5 0 7 8]; medfilt1(x)
```

Transform Functions 7

Fourier Transform Functions

This section describes the Fourier and the discrete cosine transform functions in the IPP. The IPP contains functions which perform the discrete Fourier transform (DFT), the fast Fourier transform (FFT), and the discrete cosine transform (DCT) of signal samples. It also includes variations of the basic functions to support different application requirements.

Transform Support Functions

This section describes the flag and hint arguments used by the Fourier and the discrete transform functions, the main packed formats `Perm`, `Pack`, and `CCS`, and the functions performing unpack, conversion, and multiplication of data stored in the above formats.

Flag and Hint Arguments

The Fourier transform functions require you to specify the *`flag`* and *`hint`* arguments.

The *`flag`* argument specifies the result normalization method. Table 7-1 lists the values you can enter for the *`flag`* argument. Specify one and only one of the represented values in the *`flag`* argument. The *`A`* and *`B`* factors are multipliers used in the DFT computation.

Table 7-1 Flag Arguments for Fourier Transform Functions

Value	A	B	Description
IPP_FFT_DIV_FWD_BY_N	1/*N*	1	Forward transform is done with the 1/*N* normalization.
IPP_FFT_DIV_INV_BY_N	1	1/*N*	Inverse transform is done with the 1/*N* normalization.
IPP_FFT_DIV_BY_SQRTN	$1/N^{1/2}$	$1/N^{1/2}$	Forward and inverse transform is done with the $1/N^{1/2}$ normalization.
IPP_FFT_NODIV_BY_ANY	1	1	Forward or inverse transform is done without the 1/*N* or $1/N^{1/2}$ normalization.

The *hint* argument suggests using special code, faster but less accurate calculation, or more accurate but slower calculation. Table 7-2 lists values you can enter for the *hint* argument.

Note that the *hint* argument is also used for tone generation, see "Tone-Generating Functions" on page 4-5, and for logarithmic addition, see "Model Evaluation" on page 8-59.

Table 7-2 Hint Arguments for Fourier Transform Functions

Value	Description
ippAlgHintNone	No suggestions. The function selects an algorithm
ippAlgHintFast	Fast algorithm is suggested for computation.
ippAlgHintAccurate	Accurate algorithm is suggested for computation.

Pack Format

The Pack format is a convenient, compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real FFT algorithms ("natural" in the sense that bit-reversed order is natural for radix-2 complex FFTs). In Pack format, the output samples of the FFT are arranged as shown in Table 7-3. The output signal can be unpacked to a complex signal using the function ippsConjPack, see "ConjPack" on page 7-6 for more information.

Perm Format

The Perm format stores the values in the order in which the FFT algorithm uses them. This is the most natural way of storing values for the FFT algorithm. The Perm format is an arbitrary permutation of the Pack format. An important characteristic of the Perm format is that the real and imaginary parts of a given sample need not be adjacent.

In Perm format, the output samples of the FFT are arranged as shown in Table 7-3. The output signal can be unpacked to a complex signal using the function ippsConjPerm, see "ConjPerm" on page 7-4 for more information.

CCS Format

The CCS format stores the values of the first half of the output complex signal resulted from the forward FFT.

In CCS format, the output samples of the FFT are arranged as shown in Table 7-3. Note that the signal stored in CCS format is one complex element longer. The output signal can be unpacked to a complex signal using the function ippsConjCCS, see "ConjCCS" on page 7-8 for more information.

Table 7-3 Forward FFT Result Representation in Pack, Perm, and CCS Formats

FFTReal	0	1	2	3	...	N-2	N-1	N	N+1
Pack	R_0	R_1	I_1	R_2	. . .	$I_{N/2-1}$	$R_{N/2}$		
Perm	R_0	$R_{N/2}$	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$		
CCS	R_0	0	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$	$R_{N/2}$	0

Unpack of Packed Data

The following functions ConjPerm, ConjPack, and ConjCCS convert data from the packed formats to a usual complex data format using the FFT symmetry property for transforming real data. The output data is complex, the output array length is defined by the number of complex elements in the output vector. Note that the output array size is two times as big as the input array size. The data stored in CCS format require a bigger array than the other formats. Even and odd length arrays have some specific features discussed for each function separately.

ConjPerm

Converts the data in `Perm` *format to complex data format.*

```
IppStatus ippsConjPerm_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjPerm_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjPerm_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjPerm_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjPerm_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPerm_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector (for the in-place operation).
lenDst	Number of elements in the vector.

Discussion

The function `ippsConjPerm` converts the data in `Perm` format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function `ippsConjPerm` converts the data in `Perm` format in the vector *pSrcDst* to complex data format and stores the results in *pSrcDst*.

Table 7-4 shows the examples of unpack from the `Perm` format. The `Data` column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the `Packed` column. The output result is the complex data vector in the `Extended` column. The number of vector elements is in the `Length` column. Example 7-1 shows how to use the function `ippsConjPerm`.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcDst*, *pDst*, or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *lenDst* is less than or equal to 0.

Table 7-4 Examples of Packed Data Obtained by FFT

Data	Packed	Extended	Length
FFT([1])	1	{1, 0}	1
FFT([1 2])	3, -1	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, -7, -2, 7	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Example 7-1 Using the ippsConjPerm Function

```
void ConjPerm(void) {
      Ipp16s x[8] = {1,2,3,5,6,7,8,9};
      Ipp16sc zero={0,0}, y[6];
      IppStatus st;
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjPerm_16sc( x, y, 6 );
      printf_16sc("Perm 6:", y, 6, st );
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjPerm_16sc( x, y, 5 );
      printf_16sc("Perm 5:", y, 5, st );
}
Output:
    Perm 6:  {1,0} {3,5} {6,7} {2,0} {6,-7} {3,-5}
    Perm 5:  {1,0} {2,3} {5,6} {5,-6} {2,-3}
```

ConjPack

Converts the data in `Pack` *format to complex data format.*

```
IppStatus ippsConjPack_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjPack_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjPack_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjPack_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjPack_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPack_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector (for the in-place operation).
lenDst	Number of elements in the vector.

Discussion

The function `ippsConjPack` converts the data in `Pack` format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function `ippsConjPack` converts the data in `Pack` format in the vector *pSrcDst* to complex data format and stores the results in *pSrcDst*.

Table 7-5 shows the examples of unpack from the `Pack` format. The `Data` column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the `Packed` column. The output result is the complex data vector in the `Extended` column. The number of vector elements is in the `Length` column. Example 7-2 shows how to use the function `ippsConjPack`.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcDst*, *pDst*, or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *lenDst* is less than or equal to 0.

Table 7-5 Examples of Unpack from the Pack Format

Data	Packed	Extended	Length
FFT([1])	1	{1, 0}	1
FFT([1 2])	3, -1	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, -2, 7, -7	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Example 7-2 Using the ippsConjPack Function

```
void ConjPack(void) {
      Ipp16s x[8] = {1,2,3,5,6,7,8,9};
      Ipp16sc zero={0,0}, y[6];
      IppStatus st;
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjPack_16sc( x, y, 6 );
      printf_16sc("pack 6", y, 6, st );
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjPack_16sc( x, y, 5 );
      printf_16sc("pack 5", y, 5, st );
}
Output:
    Pack 6: {1,0} {2,3} {5,6} {7,0} {5,-6} {2,-3}
    Pack 5: {1,0} {2,3} {5,6} {5,-6} {2,-3}
```

ConjCCS

Converts the data in CCS *format to complex data format.*

```
IppStatus ippsConjCCS_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjCCS_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjCCS_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjCCS_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjCCS_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjCCS_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector (for the in-place operation).
lenDst	Number of elements in the vector.

Discussion

The function ippsConjCCS converts the data in CCS format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function ippsConjCcs converts the data in CCS format in the vector *pSrcDst* to complex data format and stores the results in *pSrcDst*.

Table 7-6 shows the examples of unpack from the CCS format. The Data column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the Packed column. The output result is the complex data vector in the Extended column. The number of vector elements is in the Length column.The data stored in CCS format are two real elements longer.Example 7-3 shows how to use the function ippsConjCCS.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcDst`*, *`pDst`*, or *`pSrc`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`lenDst`* is less than or equal to 0.

Table 7-6 Examples of Unpack from the CCS Format

Data	Packed	Extended	Length
FFT([1])	1, 0	{1, 0}	1
FFT([1 2])	3, 0, -1, 0	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, 0, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, 0, -2, 7, -7, 0	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Example 7-3 Using the ippsConjCCS Function

```
void ConjCCS(void) {
      Ipp16s x[8] = {1,2,3,5,6,7,8,9};
      Ipp16sc zero={0,0}, y[6];
      IppStatus st;
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjCCS_16sc( x, y, 6 );
      printf_16sc("CCS 6:", y, 6, st );
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjCCS_16sc( x, y, 5 );
      printf_16sc("CCS 5:", y, 5, st );
}
Ouput:
    CCS 6:  {1,2} {3,5} {6,7} {8,9} {6,-7} {3,-5}
    CCS 5:  {1,2} {3,5} {6,7} {6,-7} {3,-5}
```

Multiplication of Packed Data

The functions described in this section perform the element-wise complex multiplication of vectors stored in Pack or Perm formats. These functions are used with the function ippsFFTFwd_RToPack and ippsFFTInv_PackToR to perform fast convolution on real signals.

The standard vector multiplication function ippsMul can not be used to multiply Pack or Perm format vectors because:

- Two real samples are stored in Pack format.
- The Perm format might not pair the real parts of a signal with their corresponding imaginary parts.

The argument *order* indicates base-2 logarithm of the length N of the FFT, where $N = 2^{order}$.

MulPack, MulPerm

Multiply the elements of two vectors stored in Pack *or* Perm *format.*

```
IppStatus ippsMulPack_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
      Ipp32f* pDst, int length);
IppStatus ippsMulPack_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
      Ipp64f* pDst, int length);
IppStatus ippsMulPack_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp16s* pDst, int length, int scaleFactor);
IppStatus ippsMulPack_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst,
      int length);
IppStatus ippsMulPack_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst,
      int length);
IppStatus ippsMulPack_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst,
      int length, int scaleFactor);
```

```
IppStatus ippsMulPerm_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
     Ipp32f* pDst, int length);
IppStatus ippsMulPerm_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
     Ipp64f* pDst, int length);
IppStatus ippsMulPerm_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
     Ipp16s* pDst, int length, int scaleFactor);
IppStatus ippsMulPerm_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst,
     int length);
IppStatus ippsMulPerm_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst,
     int length);
IppStatus ippsMulPerm_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst,
     int length, int scaleFactor);
```

Arguments

pSrc1, pSrc2	Pointers to the vectors whose elements are to be multiplied together.
pDst	Pointer to the destination vector which stores the result of the multiplication *pSrc1[n] * pSrc2[n]*.
pSrc	Pointer to the vector whose elements are to be multiplied by the elements of *pSrcDst* in-place.
pSrcDst	Pointer to the source and destination vector (for the in-place operation).
length	Number of elements in the vector.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The functions `ippsMulPack` and `ippsMulPerm` multiply the elements of the vector *pSrc1* by the elements of the vector *pSrc2*, and store the result in *pDst*.

The in-place functions `ippsMulPack` and `ippsMulPerm` multiply the elements of the vector *pSrc* by the elements of the vector *pSrcDst*, and store the result in *pSrcDst*.

The functions multiply the packed data according to their packed format. The data in `Perm` and `Pack` packed formats include several real values, the rest are complex. Thus, the function performs several real multiplication operations on real elements and

complex multiplication operations on complex data. Such kind of packed data multiplication is usually used for signals filtering with the FFT transform when the element-wise multiplication is performed in the frequency domain.

ippsMulPack. The function ippsMulPack performs multiplication of the data stored in Pack format.

ippsMulPerm. The function ippsMulPerm performs multiplication of the data stored in Perm format.

The vectors stored in CCS format can be multiplied using the standard function for complex data multiplication. For the functions with the Sfs suffixes scaling is performed in accordance with the *scaleFactor* value. When the output value exceeds the data range, the result may become saturated.

Example 7-4 shows how to use the function ippsMulPack_32f_I.

Return Value

ippStsNoErr	Indicates no error
ippStsNullPtrErr	Indicates an error when the *pSrcDst*, *pDst*, *pSrc1*, *pSrc2*, or *pSrc* pointer is NULL
ippStsSizeErr	Indicates an error when *length* is less than or equal to 0

Example 7-4 Using the ippsMulPack Function

```
void mulpack( void ) {
     Ipp32f x[8], X[8], h[8]={1.0f/3,1.0f/3,1.0f/3,0,0,0,0,0}, H[8];
     IppStatus st;
     IppsFFTSpec_R_32f* spec;
     st = ippsFFTInitAlloc_R_32f(&spec, 3, IPP_FFT_DIV_INV_BY_N,
         ippAlgHintNone);
     ippsSet_32f( 3, x, 8 );
     x[3] = 5;
     st = ippsFFTFwd_RToPack_32f( x, X, spec, NULL );
     st = ippsFFTFwd_RToPack_32f( h, H, spec, NULL );
     ippsMulPack_32f_I( H, X, 8 );
     st = ippsFFTInv_PackToR_32f( X, x, spec, NULL );
     printf_32f("filtered =", x, 8, st );
     ippsFFTFree_R_32f( spec );
}
Output:
    filtered =  3.0 3.0 3.0 3.666667 3.666667 3.666667 3.0 3.0
Matlab* analog:
    >> x=3*ones(1,8); x(4)=5;h=zeros(1,8); h(1:3)=1/3;
    real(ifft(fft(x).*fft(h)))
```

MulPackConj

Multiplies elements of a vector by the elements of a complex conjugate vector stored in Pack *format.*

```
IppStatus ippsMulPackConj_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst,
     int length);
IppStatus ippsMulPackConj_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst,
     int length);
```

Arguments

pSrc	Pointer to the first source vector.
pSrcDst	Pointer to the second source and destination vector .
length	Number of elements in the vector.

Discussion

The function ippsMulPackConj multiplies the elements of a source vector *pSrc* by elements of the vector that is complex conjugate to the source vector *pSrcDst* and stores the results in *pSrcDst*. The function performs only in-place operations on data stored in Pack format.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *length* is less than or equal to 0.

Fast Fourier Transform Functions

The functions described in this section compute the forward and inverse fast Fourier transform of real and complex signals. The FFT is similar to the discrete Fourier transform (DFT) but is significantly faster. The length of the vector transformed by the FFT must be a power of 2.

To use the FFT functions, initialize the specification structure which contains such data as tables of twiddle factors. The initialization functions create the specifications for both forward and inverse transforms. The amount of prior calculations is thus reduced and the overall performance increased.

The *hint* argument, passed to the initialization functions, suggests using special algorithm, faster or more accurate. The *flag* argument specifies the result normalization method. The complex signal can be represented as a single array containing complex elements, or two separate arrays containing real and imaginary parts. The output result of the FFT can be packed in `Perm`, `Pack`, or `CCS` format.

You can speed up the FFT on Intel® processors with SIMD instructions, if assign an external buffer. The external buffer increases performance because it allows to avoid allocation and deallocation of internal buffers and to store data in cache.

FFTInitAlloc_C, FFTInitAlloc_R

Initializes the fast Fourier transform specification for real and complex signals.

```
IppStatus ippsFFTInitAlloc_R_16s(IppsFFTSpec_R_16s** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_16s(IppsFFTSpec_C_16s** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_16sc(IppsFFTSpec_C_16sc** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);


IppStatus ippsFFTInitAlloc_R_32f(IppsFFTSpec_R_32f** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
```

```
IppStatus ippsFFTInitAlloc_C_32f(IppsFFTSpec_C_32f** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_32fc(IppsFFTSpec_C_32fc** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_R_64f(IppsFFTSpec_R_64f** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_64f(IppsFFTSpec_C_64f** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_64fc(IppsFFTSpec_C_64fc** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
```

Arguments

flag	Specifies the result normalization method. The values for the *flag* argument are described in "Flag and Hint Arguments."
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
order	FFT order. The input signal length is $N=2^{order}$.
pFFTSpec	Pointer to the FFT specification structure to be created.

Discussion

The function ippsFFTInitAlloc_C and ippsFFTInitAlloc_R create and initialize the FFT specification structure *pFFTSpec* with the following parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. The *order* argument defines the transform's length. Thus the input and output signals are 2^{order}-length arrays.

ippsFFTInitAlloc_C. The function ippsFFTInitAlloc_C initializes the complex FFT specification structure.

ippsFFTInitAlloc_R. The function ippsFFTInitAlloc_R initializes the real FFT specification structure.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pFFTSpec* pointer is NULL.
ippStsFftOrderErr	Indicates an error when the *order* value is incorrect.
ippStsFftFlagErr	Indicates an error when the *flag* value is incorrect.
ippStsMemAllocErr	Indicates an error when no memory allocated.

FFTFree_C, FFTFree_R

Closes a fast Fourier transform specification for real and complex signals.

```
IppStatus ippsFFTFree_R_16s(IppsFFTSpec_R_16s* pFFTSpec);
IppStatus ippsFFTFree_C_16s(IppsFFTSpec_C_16s* pFFTSpec);
IppStatus ippsFFTFree_C_16sc(IppsFFTSpec_C_16sc* pFFTSpec);

IppStatus ippsFFTFree_R_32f(IppsFFTSpec_R_32f* pFFTSpec);
IppStatus ippsFFTFree_C_32f(IppsFFTSpec_C_32f* pFFTSpec);
IppStatus ippsFFTFree_C_32fc(IppsFFTSpec_C_32fc* pFFTSpec);

IppStatus ippsFFTFree_R_64f(IppsFFTSpec_R_64f* pFFTSpec);
IppStatus ippsFFTFree_C_64f(IppsFFTSpec_C_64f* pFFTSpec);
IppStatus ippsFFTFree_C_64fc(IppsFFTSpec_C_64fc* pFFTSpec);
```

Arguments

pFFTSpec	Pointer to the FFT specification structure to be closed.

Discussion

The function ippsFFTFree closes the FFT specification structure *pFFTSpec* by freeing all memory associated with the specification created by ippsFFTInit_C or ippsFFTInit_R. Call ippsFFTFree after the transform is completed.

ippsFFTFree_C. The function ippsFFTFree_C closes the complex FFT specification structure.

ippsFFTFree_R. The function ippsFFTFree_R closes the real FFT specification structure.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pFFTSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pFFTSpec* is incorrect.

FFTGetBufSize_C, FFTGetBufSize_R

Gets the size of the FFT work buffer in bytes.

```
IppStatus ippsFFTGetBufSize_R_16s(const IppsFFTSpec_R_16s* pFFTSpec,
     int* pSize);
IppStatus ippsFFTGetBufSize_C_16s(const IppsFFTSpec_C_16s* pFFTSpec,
     int* pSize);
IppStatus ippsFFTGetBufSize_C_16sc(const IppsFFTSpec_C_16sc* pFFTSpec,
          int* pSize);

IppStatus ippsFFTGetBufSize_R_32f(const IppsFFTSpec_R_32f* pFFTSpec,
     int* pSize);
IppStatus ippsFFTGetBufSize_C_32f(const IppsFFTSpec_C_32f* pFFTSpec,
     int* pSize);
IppStatus ippsFFTGetBufSize_C_32fc(const IppsFFTSpec_C_32fc* pFFTSpec,
     int* pSize);
```

```
IppStatus ippsFFTGetBufSize_R_64f(const IppsFFTSpec_R_64f* pFFTSpec,
     int* pSize);
IppStatus ippsFFTGetBufSize_C_64f(const IppsFFTSpec_C_64f* pFFTSpec,
     int* pSize);
IppStatus ippsFFTGetBufSize_C_64fc(const IppsFFTSpec_C_64fc* pFFTSpec,
     int* pSize);
```

Arguments

pFFTSpec	Pointer to the FFT specification structure.
pSize	Pointer to the FFT work buffer size value.

Discussion

The function ippsFFTGetBufSize gets the work buffer size of the FFT described by the specification structure *pFFTSpec* in bytes and stores in *pSize*.

ippsFFTGetBufSize_C. The function ippsFFTGetBufSize_C gets the size of the complex FFT work buffer.

ippsFFTGetBufSize_R. The function ippsFFTGetBufSize_R gets the size of the real FFT work buffer.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pFFTSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pFFTSpec* is incorrect.

FFTFwd_CToC, FFTInv_CToC

Computes the forward or inverse fast Fourier transform (FFT) of a complex signal.

```
IppStatus ippsFFTFwd_CToC_32f(const Ipp32f* pSrcRe,
      const Ipp32f* pSrcIm, Ipp32f* pDstRe, Ipp32f* pDstIm,
      const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_32f(const Ipp32f* pSrcRe,
      const Ipp32f* pSrcIm, Ipp32f* pDstRe, Ipp32f* pDstIm,
      const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_64f(const Ipp64f* pSrcRe,
      const Ipp64f* pSrcIm, Ipp64f* pDstRe, Ipp64f* pDstIm,
      const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_64f(const Ipp64f* pSrcRe,
      const Ipp64f* pSrcIm, Ipp64f* pDstRe, Ipp64f* pDstIm,
      const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      const IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      const IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
      const IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
      const IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_CToC_16s_Sfs(const Ipp16s* pSrcRe,
      const Ipp16s* pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm,
      const IppsFFTSpec_C_16s* pFFTSpecx, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_16s_Sfs(const Ipp16s* pSrcRe,
      const Ipp16s* pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm,
      const IppsFFTSpec_C_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     const IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     const IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Arguments

pFFTSpec	Pointer to the FFT specification structure.
pSrc	Pointer to the input array containing complex values.
pDst	Pointer to the output array containing complex values.
pSrcRe	Pointer to the input array containing real parts of the signal.
pSrcIm	Pointer to the input array containing imaginary parts of the signal.
pDstRe	Pointer to the output array containing real parts of the signal.
pDstIm	Pointer to the output array containing imaginary parts of the signal.
pBuffer	Pointer to the FFT work buffer.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The functions `ippsFFTFwd_CToC` and `ippsFFTInv_CToC` compute the forward or inverse FFT of a complex signal according to the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*.

The functions using the complex data type, e.g., with the `32fc` suffixes, process the input complex array *pSrc* and store the result in *pDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, e.g., with the `32f` suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained. The *pBuffer* argument provides the FFT functions with the necessary working memory and allows to avoid

memory allocation within the functions. The buffer allows to increase performance if the FFT functions use the result of the previous operation stored in cache as an input array. The buffer is same for both consecutive operations.

ippsFFTFwd_CToC. The function ippsFFTFwd_CToC computes a complex forward FFT. The length of the FFT must be a power of 2.

ippsFFTInv_CToC. The function ippsFFTInv_CToC computes a complex inverse FFT. The length of the FFT must be a power of 2.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pFFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory allocated.

FFTFwd_RToPerm, FFTInv_PermToR, FFTFwd_RToPack, FFTInv_PackToR, FFTFwd_RToCCS, FFTInv_CCSToR

Computes the forward or inverse fast Fourier transform (FFT) of a real signal.

```
IppStatus ippsFFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPerm_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CCSToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Arguments

pFFTSpec	Pointer to the FFT specification structure.
pSrc	Pointer to the input array containing real values for the forward transform and packed complex values resulted from the inverse transform.

pDst	Pointer to the output array containing real values for the inverse transform and packed complex values resulted from the forward transform.
pBuffer	Pointer to the work buffer.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

These functions compute the forward or inverse FFT of a real signal according to the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*.

The result of the forward transform (i.e., in the frequency-domain) of real signals is represented in several possible packed formats: Pack, Perm, or CCS. The data can be packed due to the symmetry property of the FFT transform of a real signal.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained. The *pBuffer* argument provides the FFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the FFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations.

ippsFFTFwd_RToPerm, ippsFFTInv_PermToR. These functions compute the forward or inverse FFT and store the result in Perm format. The length of the FFT must be a power of 2.

ippsFFTFwd_RToPack, ippsFFTInv_PackToR. These functions compute the forward or inverse FFT and store the result in Pack format. The length of the FFT must be a power of 2.

ippsFFTFwd_RToCCS, ippsFFTInv_CCSToR. These functions compute the forward or inverse FFT and store the result in CCS format. The length of the FFT must be a power of 2.

Table 7-7 shows how the output result is represented in each of the packed format: Pack, Perm, and CCS. Example 7-5 shows how to initialize the specification and call ippsFFTFwd_RToCCS_32f.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pFFTSpec* is incorrect.
`ippStsMemAllocErr`	Indicates an error when no memory allocated.

Table 7-7 Forward FFT Result Representation in Pack, Perm, and CCS Formats

FFTReal	**0**	**1**	**2**	**3**	**...**	**N-2**	**N-1**	**N**	**N+1**
Pack	R_0	R_1	I_1	R_2	. . .	$I_{N/2-1}$	$R_{N/2}$		
Perm	R_0	$R_{N/2}$	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$		
CCS	R_0	0	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$	$R_{N/2}$	0

Example 7-5 Using the ippsFFTFwd_RToCCS Function

```
IppStatus fft( void ) {
     Ipp32f x[8], X[10];
     int n;
     IppStatus status;
     IppsFFTSpec_R_32f* spec;
     status = ippsFFTInitAlloc_R_32f(&spec, 3, IPP_FFT_DIV_INV_BY_N,
         ippAlgHintNone );
     for(n=0; n<8; ++n) x[n] = (float)cos(IPP_2PI *n *16/64);
     status = ippsFFTFwd_RToCCS_32f( x, X, spec, NULL );
     ippsMagnitude_32fc( (Ipp32fc*)X, x, 4 );
     ippsFFTFree_R_32f( spec );
     printf_32f("fft magn =", x, 4, status );
     return status;
}
Output:
    fft magn = 0.000000 0.000000 4.000000 0.000000
Matlab* Analog:
    >> n=0:7; x=sin(2*pi*n*16/64); X=abs(fft(x)); X(1:4)
```

Discrete Fourier Transform Functions

The functions described in this section compute the forward and inverse discrete Fourier transform of real and complex signals. The DFT is less efficient than the fast Fourier transform but the length of the vector transformed by the DFT can be arbitrary.

The *hint* argument, passed to the initialization functions, suggests using special algorithm, faster or more accurate. The *flag* argument specifies the result normalization method. The complex signal can be represented as a single array containing complex elements, or two separate arrays containing real and imaginary parts. The output result of the FFT can be packed in `Perm`, `Pack`, or `CCS` format.

To use the DFT functions, initialize the specification structure which contains such data as tables of twiddle factors. The initialization functions create the specifications for both forward and inverse transforms.

For more information about the fast computation of the discrete Fourier transform, see [Mit93], section 8-2, *Fast Computation of the DFT.*

DFTInitAlloc_C, DFTInitAlloc_R

Initializes the discrete Fourier transform specification for real and complex signals.

```
IppStatus ippsDFTInitAlloc_R_16s(IppsDFTSpec_R_16s** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_16s(IppsDFTSpec_C_16s** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_16sc(IppsDFTSpec_C_16sc** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);

IppStatus ippsDFTInitAlloc_R_32f(IppsDFTSpec_R_32f** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_32f(IppsDFTSpec_C_32f** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_32fc(IppsDFTSpec_C_32fc** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);

IppStatus ippsDFTInitAlloc_R_64f(IppsDFTSpec_R_64f** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_64f(IppsDFTSpec_C_64f** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_64fc(IppsDFTSpec_C_64fc** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
```

Arguments

flag	Specifies the result normalization method. The values for the *flag* argument are described in "Flag and Hint Arguments."
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
length	Length of the DFT transform.
pDFTSpec	Pointer to the DFT specification structure to be created.

Discussion

The functions ippsDFTInitAlloc_C and ippsDFTInitAlloc_R create and initialize the DFT specification structure *pDFTSpec* with the following parameters: the transform *length*, the normalization *flag*, and the specific code *hint*. The *length* argument defines the transform's length.

ippsDFTInitAlloc_C. The function ippsDFTInitAlloc_C initializes the complex DFT specification structure.

ippsDFTInitAlloc_R. The function ippsDFTInitAlloc_R initializes the real DFT specification structure.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDFTSpec* pointer is NULL.
ippStsFftOrderErr	Indicates an error when the *order* value is incorrect.
ippStsFftFlagErr	Indicates an error when the *flag* value is incorrect.
ippStsMemAllocErr	Indicates an error when no memory allocated.

DFTFree_C, DFTFree_R

Closes the discrete Fourier transform specification for real and complex signals.

```
IppStatus ippsDFTFree_R_16s(IppsDFTSpec_R_16s* pDFTSpec);
IppStatus ippsDFTFree_C_16s(IppsDFTSpec_C_16s* pDFTSpec);
IppStatus ippsDFTFree_C_16sc(IppsDFTSpec_C_16sc* pDFTSpec);

IppStatus ippsDFTFree_R_32f(IppsDFTSpec_R_32f* pDFTSpec);
IppStatus ippsDFTFree_C_32f(IppsDFTSpec_C_32f* pDFTSpec);
IppStatus ippsDFTFree_C_32fc(IppsDFTSpec_C_32fc* pDFTSpec);

IppStatus ippsDFTFree_R_64f(IppsDFTSpec_R_64f* pDFTSpec);
IppStatus ippsDFTFree_C_64f(IppsDFTSpec_C_64f* pDFTSpec);
IppStatus ippsDFTFree_C_64fc(IppsDFTSpec_C_64fc* pDFTSpec);
```

Arguments

pDFTSpec	Pointer to the DFT specification structure to be closed.

Discussion

The function ippsDFTFree closes the DFT specification structure *pDFTSpec* by freeing all memory associated with the specification created by ippsDFTInitAlloc_C or ippsDFTInitAlloc_R. Call ippsDFTFree after the transform is completed.

ippsDFTFree_C. The function ippsDFTFree_C closes the complex DFT specification structure.

ippsDFTFree_R. The function ippsDFTFree_R closes the real DFT specification structure.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pDFTSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.

DFTGetBufSize_C, DFTGetBufSize_R

Gets the size of the DFT work buffer in bytes.

```
IppStatus ippsDFTGetBufSize_R_16s(const IppsDFTSpec_R_16s*  pDFTSpec,
      int* pSize);
IppStatus ippsDFTGetBufSize_C_16s(const IppsDFTSpec_C_16s*  pDFTSpec,
      int* pSize);
IppStatus ippsDFTGetBufSize_C_16sc(const IppsDFTSpec_C_16sc* pDFTSpec,
      int* pSize);

IppStatus ippsDFTGetBufSize_R_32f(const IppsDFTSpec_R_32f*  pDFTSpec,
      int* pSize);
IppStatus ippsDFTGetBufSize_C_32f(const IppsDFTSpec_C_32f*  pDFTSpec,
      int* pSize);
IppStatus ippsDFTGetBufSize_C_32fc(const IppsDFTSpec_C_32fc* pDFTSpec,
      int* pSize);

IppStatus ippsDFTGetBufSize_R_64f(const IppsDFTSpec_R_64f*  pDFTSpec,
      int* pSize);
IppStatus ippsDFTGetBufSize_C_64f(const IppsDFTSpec_C_64f*  pDFTSpec,
      int* pSize);
IppStatus ippsDFTGetBufSize_C_64fc(const IppsDFTSpec_C_64fc* pDFTSpec,
      int* pSize);
```

Arguments

pDFTSpec	Pointer to the DFT specification structure.
pSize	Pointer to the DFT work buffer size value.

Discussion

The function ippsDFTGetBufSize gets the work buffer size of the DFT described by the specification structure *pDFTSpec* in bytes and stores in *pSize*.

ippsDFTGetBufSize_C. The function ippsDFTGetBufSize_C gets the size of the complex DFT work buffer.

ippsDFTGetBufSize_R. The function ippsDFTGetBufSize_R gets the size of the real DFT work buffer.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDFTSpec* is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.

DFTFwd_CToC, DFTInv_CToC

Computes the forward or inverse discrete Fourier transform (DFT) of a complex signal.

```
IppStatus ippsDFTFwd_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f*
     pSrcIm, Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f*
     pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f*
     pSrcIm, Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f*
     pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f*
     pSrcIm, Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f*
     pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f*
     pSrcIm, Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f*
     pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     const IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     const IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     const IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     const IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s*
     pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsDFTSpec_C_16s*
     pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s*
     pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsDFTSpec_C_16s*
     pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     const IppsDFTSpec_C_16sc* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     const IppsDFTSpec_C_16sc* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Arguments

pDFTSpec	Pointer to the DFT specification structure.
pSrc	Pointer to the input array containing complex values.
pDst	Pointer to the output array containing complex values.
pSrcRe	Pointer to the input array containing real parts of the signal.
pSrcIm	Pointer to the input array containing imaginary parts of the signal.
pDstRe	Pointer to the output array containing real parts of the signal.
pDstIm	Pointer to the output array containing imaginary parts of the signal.
pBuffer	Pointer to the work buffer.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The functions ippsDFTFwd_CToC and ippsDFTInv_CToC compute the forward or inverse DFT of a complex signal according to the *pDFTSpec* specification parameters: the transform *length*, the normalization *flag*, and the specific code *hint*.

The functions using the complex data type, e.g., with 32fc suffixes, process the input complex array *pSrc* and store the result in *pDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, e.g., with 32f suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained. The *pBuffer* argument provides the DFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations.

The forward DFT functionality can be described as follows:

$$X(k) = A \sum_{n=0}^{N-1} x(n) \cdot exp\left(-j2\pi\frac{kn}{N}\right),$$

where k is the index of elements in the frequency domain, n is the index of elements in the time domain, N is the input signal *length*, and A and B are multipliers defined by *flag*. Also, in the forward direction, $x(n)$ is *pSrc[n]* and $X(k)$ is *pDst[k]*; in the inverse direction, $x(n)$ is *pDst[n]* and $X(k)$ is *pSrc[k]*.

The definition of the inverse discrete Fourier transform is:

$$x(k) = B \sum_{n=0}^{N-1} X(n) \cdot exp\left(j2\pi\frac{kn}{N}\right)$$

ippsDFTFwd_CToC. The function ippsDFTFwd_CToC computes the complex forward DFT.

ippsDFTInv_CToC. The function ippsDFTInv_CToC computes the complex inverse DFT.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *`pDFTSpec`* is incorrect.
`ippStsMemAllocErr`	Indicates an error when no memory allocated.

DFTFwd_RToPerm, DFTInv_PermToR, DFTFwd_RToPack, DFTInv_PackToR, DFTFwd_RToCCS, DFTInv_CCSToR

Computes the forward or inverse fast Fourier transform (DFT) of a real signal.

```
IppStatus ippsDFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTFwd_RToPerm_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPack_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToCCS_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PermToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PackToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus, ippsDFTInv_CCSToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Arguments

pDFTSpec	Pointer to the DFT specification structure.
pSrc	Pointer to the input array containing complex values for the forward transform and packed complex values resulted from the inverse transform.
pDst	Pointer to the output array containing real values for the inverse transform and packed complex values resulted from the forward transform.
pBuffer	Pointer to the work buffer.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

These functions compute the forward or inverse DFT of a real signal according to the *pDFTSpec* specification parameters: the transform *length*, the normalization *flag*, and the specific code *hint*.

The result of the forward transform (i.e. in the frequency-domain) of real signals is represented in several possible packed formats: `Pack`, `Perm`, or `CCS`. The data can be packed due to the symmetry property of the DFT transform of a real signal.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained. The *pBuffer* argument provides the DFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations.

The forward DFT functionality can be described as follows:

$$X(k) = A \sum_{n=0}^{N-1} x(n) \cdot exp\left(-j2\pi\frac{kn}{N}\right),$$

where k is the index of elements in the frequency domain, n is the index of elements in the time domain, N is the input signal *length*, and A and B are multipliers defined by *flag*. Also, in the forward direction, $x(n)$ is *pSrc[n]* and $X(k)$ is *pDst[k]*; in the inverse direction, $x(n)$ is *pDst[n]* and $X(k)$ is *pSrc[k]*.

The definition of the inverse discrete Fourier transform is:

$$x(k) = B \sum_{n=0}^{N-1} X(n) \cdot exp\left(j2\pi\frac{kn}{N}\right)$$

ippsDFTFwd_RToPerm, ippsDFTInv_PermToR. These functions compute the forward or inverse DFT and store the result in `Perm` format.

ippsDFTFwd_RToPack, ippsDFTInv_PackToR. These functions compute the forward or inverse DFT and store the result in `Pack` format.

ippsDFTFwd_RToCCS, ippsDFTInv_CCSToR. These functions compute the forward or inverse DFT and store the result in `CCS` format.

Example 7-6 shows how to initialize the specification and call ippsDFTFwd_RToCCS_32f.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory allocated.

Example 7-6 Using the ippsDFTFwd_RToCCS Function

```
IppStatus dft( void ) {
     Ipp32f x[7], X[8];
     int n;
     IppStatus status;
     IppsDFTSpec_R_32f* spec;
     status = ippsDFTInitAlloc_R_32f(&spec, 7, IPP_FFT_DIV_INV_BY_N,
          ippAlgHintNone);
     for( n=0; n<7; ++n ) x[n] = (float) cos(IPP_2PI * n * 14 / 49);
     status = ippsDFTFwd_RToCCS_32f( x, X, spec, NULL );
     ippsMagnitude_32fc( (Ipp32fc*)X, x, 4 );
     ippsDFTFree_R_32f( spec );
     printf_32f("dft magn =", x, 4, status );
     return status;
}
Output:
    dft magn =  0.000000 0.000000 3.500000 0.000000
Matlab* analog:
    >> N=7;F=14/49;n=0:N-1;x=cos(2*pi*n*F);y=abs(fft(x));y(1:4)
```

DFT for a Given Frequency (Goertzel) Functions

The functions described in this section compute a single or a number of the discrete Fourier transforms for a given frequency. Note that the DFT exists only for the following normalized frequencies: 0, 1/*N*, 2/*N*,... (*N*-1)/*N*, where *N* is the number of time domain samples. Therefore you must select the frequency value from the above set.

These IPP functions use a Goertzel algorithm [Mit98] (see the Bibliography section) and are more efficient when a small number of DFT values is needed.

Some of the functions compute two values, not one. The applications computing several values, e.g., the dual-tone multifrequency signal detection, work faster, especially on Intel® processors with SIMD instructions.

Goertz

Computes the DFT for a given frequency for a single complex signal.

```
IppStatus ippsGoertz_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pVal,
      Ipp32f freq);
IppStatus ippsGoertz_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pVal,
      Ipp64f freq);
IppStatus ippsGoertz_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc*
      pVal, Ipp32f freq, int scaleFactor);
```

Arguments

freq	Pointer to the single relative frequency value [0, 1.0).
pSrc	Pointer to the input complex data vector.
len	Number of elements in the vector.
pVal	Pointer to the output DFT value.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsGoertz computes a DFT for a complex input *len*-length signal *pSrc* for a given frequency *freq*, and stores the result in *pVal*.

The ippsGoertz functionality can be described as follows:

$$y(k) = \sum_{n=0}^{N-1} x(n) \cdot exp\left(-j2\pi\frac{kn}{N}\right),$$

where *k/N* is the normalized *freq* value for which the DFT is computed.

Example 7-7 illustrates the use of Goertzel functions for selecting the magnitudes of a given frequency when computing DFTs.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsRelFreqErr	Indicates an error when *freq* is out of range.
ippStsSizeErr	Indicates an error when *len* is less or equal to 0.

Example 7-7 Using Goertzel Functions for Selecting Magnitudes of a Given Frequency

```
IppStatus goertzel( void ) {
#undef  LEN
#define LEN 100
     IppStatus status;
     Ipp32fc *x = ippsMalloc_32fc( LEN ), y;
     int n;
     ///generate a signal of 60 Hz freq that
     /// is sampled with 400 Hz freq
     for( n=0; n<LEN; ++n) {
        x[n].re =(Ipp32f)sin(IPP_2PI * n * 60 / 400);
        x[n].im = 0;
     }
     status = ippsGoertz_32fc( x, LEN, &y, 60.0f / 400 );
     printf_32fc("goertz =", &y, 1, status );
     ippsFree( x );
     return status;
}
Output:
    goertz =  {0.000090,-50.000008}
Matlab* Analog
    >> N=100;F=60/400;n=0:N-1;x=sin(2*pi*n*F);y=fft(x);n=N*F;y(n+1)
```

GoertzTwo

Computes two DFTs for a given frequency for a single complex signal.

```
IppStatus ippsGoertzTwo_32fc(const Ipp32fc* pSrc, int len,
      Ipp32fc pVal[2], const Ipp32f freq[2]);
IppStatus ippsGoertzTwo_64fc(const Ipp64fc* pSrc, int len,
      Ipp64fc pVal[2], const Ipp64f freq[2]);
IppStatus ippsGoertzTwo_16sc_Sfs(const Ipp16sc* pSrc, int len,
      Ipp16sc pVal[2], const Ipp32f freq[2], int scaleFactor);
```

Arguments

freq	Pointer to two single relative frequency value [0, 1.0).
pSrc	Pointer to the input complex data vector.
len	Number of elements in the vector.
pVal	Pointer to the output DFT values.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsGoertzTwo computes two DFTs for a complex input *len*-length signal *pSrc* for two given frequencies *freq*, and stores the result in *pVal*. The computation of two DFTs on an Intel® Pentium® III processor is performed at the same speed as one. Therefore, the applications computing several DFTs are faster.

The ippsGoertz functionality can be described as follows:

$$y(k) = \sum_{n=0}^{N-1} x(n) \cdot exp\left(-j2\pi\frac{kn}{N}\right),$$

where *k/N* is one of the normalized *freq* values for which the DFTs are computed.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsRelFreqErr	Indicates an error when *freq* is out of range.
ippStsSizeErr	Indicates an error when *len* is less or equal to 0.

Discrete Cosine Transform Functions

This section describes the functions that compute the discrete cosine transform of a signal.

DCTFwdInitAlloc, DCTInvInitAlloc

Initializes the discrete cosine transform specification.

```
IppStatus ippsDCTFwdInitAlloc_16s(IppsDCTFwdSpec_16s** pDCTSpec,
     int length, IppHintAlgorithm hint);
IppStatus ippsDCTInvInitAlloc_16s(IppsDCTInvSpec_16s** pDCTSpec,
     int length, IppHintAlgorithm hint);

IppStatus ippsDCTFwdInitAlloc_32f(IppsDCTFwdSpec_32f** pDCTSpec,
     int length, IppHintAlgorithm hint);
IppStatus ippsDCTInvInitAlloc_32f(IppsDCTInvSpec_32f** pDCTSpec,
     int length, IppHintAlgorithm hint);

IppStatus ippsDCTFwdInitAlloc_64f(IppsDCTFwdSpec_64f** pDCTSpec,
     int length, IppHintAlgorithm hint);
IppStatus ippsDCTInvInitAlloc_64f(IppsDCTInvSpec_64f** pDCTSpec,
     int length, IppHintAlgorithm hint);
```

Arguments

flag	Specifies the result normalization method. The values for the *flag* argument are described in "Flag and Hint Arguments."
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
length	Number of samples in the DCT.
pDCTSpec	Pointer to the DCT specification structure to be created.

Discussion

The function ippsDCTFwdInitAlloc and ippsDCTInvInitAlloc create and initialize the DCT specification structure *pDCTSpec* with the following parameters: the transform *length*, the normalization *flag*, and the specific code *hint*. The *length* argument defines the transform's length.

ippsDCTFwdInitAlloc. The function ippsDCTFwdInitAlloc initializes the forward DCT specification structure.

ippsDCTInvInitAlloc. The function ippsDCTInvInitAlloc initializes the inverse DCT specification structure.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDCTSpec* pointer is NULL.
ippStsSizeErr	Indicates an error when *length* is less than or equal to 0.
ippStsMemAllocErr	Indicates an error when no memory allocated.

DCTFwdFree, DCTInvFree

Closes a discrete cosine transform specification.

```
IppStatus ippsDCTFwdFree_16s(IppsDCTFwdSpec_16s* pDCTSpec);
IppStatus ippsDCTInvFree_16s(IppsDCTInvSpec_16s* pDCTSpec);
IppStatus ippsDCTFwdFree_32f(IppsDCTFwdSpec_32f* pDCTSpec);
IppStatus ippsDCTInvFree_32f(IppsDCTInvSpec_32f* pDCTSpec);
IppStatus ippsDCTFwdFree_64f(IppsDCTFwdSpec_64f* pDCTSpec);
IppStatus ippsDCTInvFree_64f(IppsDCTInvSpec_64f* pDCTSpec);
```

Arguments

pDCTSpec	Pointer to the DCT specification structure to be closed.

Discussion

The function ippsFFTFwdFree and ippsFFTInvFree close the DCT specification structure *pDCTSpec* by freeing all memory associated with the specification created by ippsDCTFwdInitAlloc or ippsDCTInvInitAlloc. Call either ippsDCTFwdFree or ippsDCTInvFree after the transform is completed.

ippsDCTFwdFree. The function ippsDCTFwdFree closes the forward DCT specification structure.

ippsDCTInvFree. The function ippsDCTInvFree closes the inverse DCT specification structure.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDCTSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDCTSpec* is incorrect.

DCTFwdGetBufSize, DCTInvGetBufSize

Gets the size of the DCT work buffer in bytes.

```
IppStatus ippsDCTFwdGetBufSize_16s(const IppsDCTFwdSpec_16s* pDCTSpec,
     int* pSize);
IppStatus ippsDCTInvGetBufSize_16s(const IppsDCTInvSpec_16s* pDCTSpec,
     int* pSize);

IppStatus ippsDCTFwdGetBufSize_32f(const IppsDCTFwdSpec_32f* pDCTSpec,
     int* pSize);
IppStatus ippsDCTInvGetBufSize_32f(const IppsDCTInvSpec_32f* pDCTSpec,
     int* pSize);

IppStatus ippsDCTFwdGetBufSize_64f(const IppsDCTFwdSpec_64f* pDCTSpec,
     int* pSize);
IppStatus ippsDCTInvGetBufSize_64f(const IppsDCTInvSpec_64f* pDCTSpec,
     int* pSize);
```

Arguments

pDCTSpec	Pointer to the DCT specification structure.
pSize	Pointer to the DCT work buffer size value.

Discussion

The functions `ippsDCTFwdGetBufSize` and `ippsDCTInvGetBufSize` get the work buffer size of the DCT described by the specification structure *pDCTSpec* in bytes and stores in *pSize*.

`ippsDCTFwdGetBufSize`. The function `ippsDCTFwdGetBufSize` gets the work buffer size of the forward DCT.

`ippsDCTInvGetBufSize`. The function `ippsDCTInvGetBufSize` gets the work buffer size of the inverse DCT.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDCTSpec* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pDCTSpec* is incorrect.

DCTFwd, DCTInv

Computes the forward or inverse discrete cosine transform (DCT) of a signal.

```
IppStatus ippsDCTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDCTFwdSpec_32f* pDCTSpec, Ipp8u* pBuffer);
IppStatus ippsDCTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDCTInvSpec_32f* pDCTSpec, Ipp8u* pBuffer);

IppStatus ippsDCTFwd_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDCTFwdSpec_64f* pDCTSpec, Ipp8u* pBuffer);
IppStatus ippsDCTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDCTInvSpec_64f* pDCTSpec, Ipp8u* pBuffer);

IppStatus ippsDCTFwd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
     IppsDCTFwdSpec_16s* pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDCTInv_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
     IppsDCTInvSpec_16s* pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Arguments

pDCTSpec	Pointer to the DCT specification structure.
pSrc	Pointer to the input data array.
pDst	Pointer to the output data array.

pBuffer	Pointer to the DCT work buffer.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsDCTFwd and ippsDCTInv compute the forward and inverse discrete cosine transform (DCT). If *length* is a power of 2, the functions use an efficient algorithm that is significantly faster than the direct computation of DCT. For other values of *length*, these functions use the direct formulas given below; however, the symmetry of the cosine function is taken into account, which allows to perform about half of the multiplication operations in the formulas.

In the following definition of DCT, $N =$ *length*,

$C(k) = \frac{1}{\sqrt{N}}$ for $k = 0$, $C(k) = \frac{\sqrt{2}}{\sqrt{N}}$ for $k > 0$;

$x(n)$ is *pSrc[n]* and $y(k)$ is *pDst[k]* for the forward DCT;

$x(n)$ is *pDst[n]* and $y(k)$ is *pSrc[k]* for the inverse DCT.

The forward DCT is defined by the formula:

$$y(k) = C(k) \sum_{n=0}^{N-1} x(n) \cdot \cos\frac{(2n+1)\pi k}{2N}$$

The definition of the inverse discrete cosine transform is:

$$x(n) = \sum_{k=0}^{N-1} C(k)y(k) \cdot \cos\frac{(2n+1)\pi k}{2N}$$

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained. The *pBuffer* argument provides the DCT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer may also increase up performance if the DCT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations.

ippsDCTFwd. The function ippsDCTFwd computes a forward DCT.

ippsDCTInv. The function ippsDCTInv computes an inverse DCT.

Example 7-8 shows how to use the functions ippsDCTFwd_32f and ippsDCTInv_32f to compress and reconstruct a signal.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pDCTSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDCTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory allocated.

Example 7-8 Using ippsDCTFwd and ippsDCTInv to Compress and Reconstruct a Signal

```
        void dct( void ) {
        #define LEN 256
             Ipp32f x[LEN], y[LEN];
             int n;
             IppsDCTFwdSpec_32f* fspec;
             IppsDCTInvSpec_32f* ispec;
             IppStatus status;
             /// data: Gaussian function, magn =1 and sigma=N/3
             for(n=0; n<LEN; ++n)
                 x[n] = (float)(exp(-0.5*(LEN/2-n)*(LEN/2-n)/(LEN/3.0)));
             /// get cosine transform coefficients
             status = ippsDCTFwdInitAlloc_32f( &fspec, LEN, ippAlgHintNone );
             status = ippsDCTFwd_32f( x, y, fspec, NULL );
             ippsDCTFwdFree_32f( fspec );
             /// Set 3/4 of these coefficients to zero
             for(n=LEN/4; n<LEN; ++n) y[n] = 0.0f;
             /// restore signal using len/4 values
             status = ippsDCTInvInitAlloc_32f( &ispec, LEN, ippAlgHintNone );
             status = ippsDCTInv_32f( y, x, ispec, NULL );
             ippsDCTInvFree_32f( ispec );
}
```

Wavelet Transform Functions

This chapter discusses the wavelet transform functions.

In signal processing, signals can represented in both frequency and time-frequency domains. In many cases the wavelet transforms become an alternative to short time Fourier transforms.

The discrete wavelet signal can be considered as a set of the coefficients $a_{i,k}$ with two indices, one of which is a “frequency” characteristic and the other is a time localization. The coefficient value corresponds to the localized wave amplitude or to one of basis transform functions. The “frequency” index shows the time scale of the localized wave. Function bases originated from one local wave by decreasing the wave by 2^n in time are the most widely used. Such transforms can be used for building very efficient implementations called fast wavelet transforms by analogy with fast Fourier transforms. Figure 7-1 shows how the time and frequency plane is divided into areas that correspond to the local wave amplitudes. This kind of transforms is implemented in the IPP and referred to as the discrete wavelet transform (DWT).

The DWT is one of the wavelet analysis methods that stem from the basis functions related to the scale factor 2. Thus, there is a basic common element shared by the DWT and the other packet analysis methods.

Likewise another basic element for signal reconstruction or synthesis can be defined, called the one-level inverse DWT. Figure 7-2 shows the diagram of the forward DWT which allows to switch to time-frequency representation shown in Figure 7-1. The diagram includes three levels of decomposition. Figure 7-3 shows the corresponding procedure of signal reconstruction based on the elementary one-level inverse transform.

The implementation of discrete multi-scale transforms is based on the use of interpolation and decimation filters with the resampling factor 2. The basis of the multi-scale signal decomposition and reconstruction functions uniquely defines the filter parameters. The IPP multi-scale transform functions use filters with finite impulse response.

The Primitives contains two sets of functions.

- Transforms designed for fixed filter banks. These transforms yield the highest performance.
- Transforms that enable the user to work with arbitrary filters. These functions use effective polyphase filtration algorithms. The transform interface gives the option of processing the data in blocks, including in real-time applications.

Figure 7-1 Wavelet Decomposition Coefficients in Time-Frequency Domain

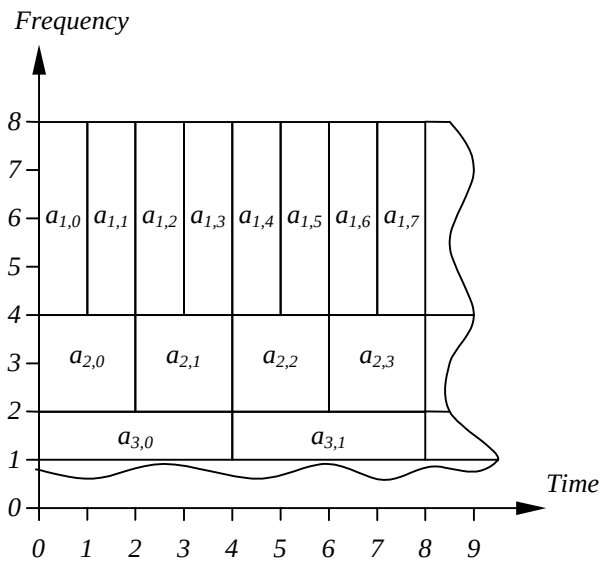


Figure 7-2 Three-Level Discrete Wavelet Decomposition

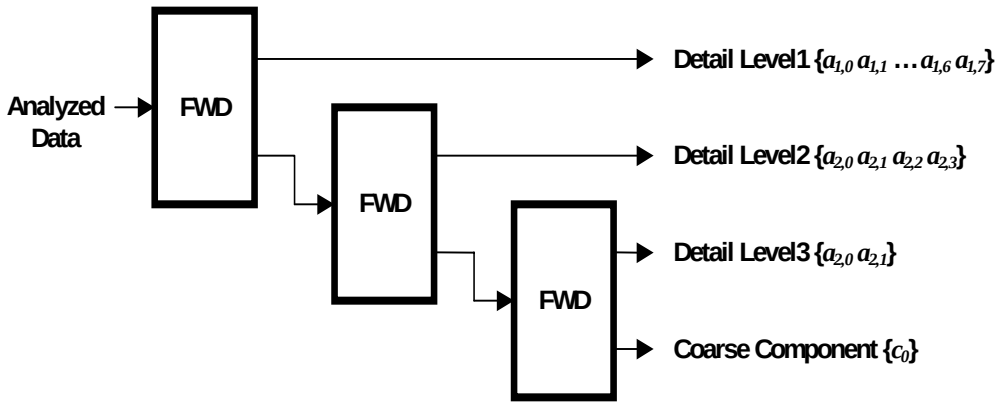
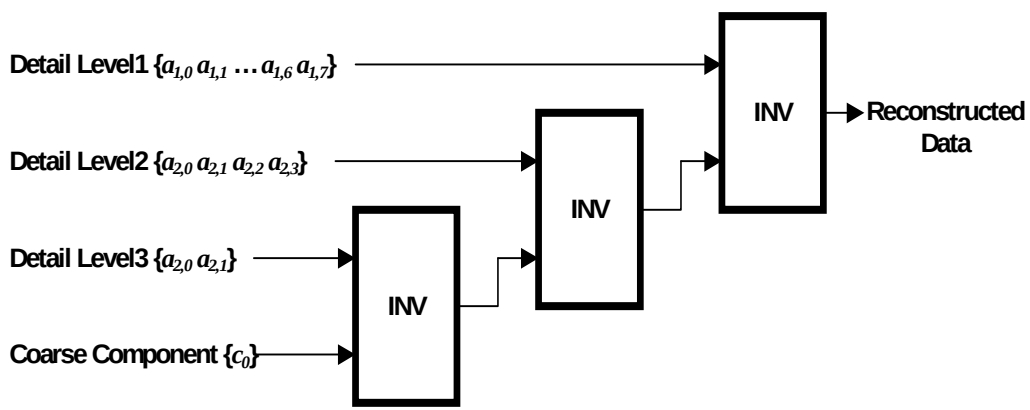


Figure 7-3 Three-Level Discrete Wavelet Reconstruction



Transforms for Fixed Filter Banks

This section describes the functions that perform forward or inverse wavelet transforms for fixed filter banks.

WTHaarFwd, WTHaarInv

Performs forward or inverse single-level discrete wavelet Haar transforms.

```
IppStatus ippsWTHaarFwd_8s(const Ipp8s* pSrc, int lenSrc,
      Ipp8s* pDstLow, Ipp8s* pDstHigh);
IppStatus ippsWTHaarFwd_16s(const Ipp16s* pSrc, int lenSrc,
      Ipp16s* pDstLow, Ipp16s* pDstHigh);
IppStatus ippsWTHaarFwd_32s(const Ipp32s* pSrc, int lenSrc,
      Ipp32s* pDstLow, Ipp32s* pDstHigh);
IppStatus ippsWTHaarFwd_64s(const Ipp64s* pSrc, int lenSrc,
      Ipp64s* pDstLow, Ipp64s* pDstHigh);
IppStatus ippsWTHaarFwd_32f(const Ipp32f* pSrc, int lenSrc,
      Ipp32f* pDstLow, Ipp32f* pDstHigh);
IppStatus ippsWTHaarFwd_64f(const Ipp64f* pSrc, int lenSrc,
      Ipp64f* pDstLow, Ipp64f* pDstHigh);
IppStatus ippsWTHaarFwd_8s_Sfs(const Ipp8s* pSrc, int lenSrc,
      Ipp8s* pDstLow, Ipp8s* pDstHigh, int scaleFactor);
IppStatus ippsWTHaarFwd_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
      Ipp16s* pDstLow, Ipp16s* pDstHigh, int scaleFactor );
IppStatus ippsWTHaarFwd_32s_Sfs(const Ipp32s* pSrc, int lenSrc,
      Ipp32s* pDstLow, Ipp32s* pDstHigh, int scaleFactor);
IppStatus ippsWTHaarFwd_64s_Sfs(const Ipp64s* pSrc, int lenSrc,
      Ipp64s* pDstLow, Ipp64s* pDstHigh, int scaleFactor);

IppStatus ippsWTHaarInv_8s(const Ipp8s* pSrcLow, const Ipp8s* pSrcHigh,
      Ipp8s* pDst, int lenDst);
```

```
IppStatus ippsWTHaarInv_16s(const Ipp16s* pSrcLow, const Ipp16s* pSrcHigh,
     Ipp16s* pDst, int lenDst);
IppStatus ippsWTHaarInv_32s(const Ipp32s* pSrcLow, const Ipp32s* pSrcHigh,
     Ipp32s* pDst, int lenDst);
IppStatus ippsWTHaarInv_64s(const Ipp64s* pSrcLow, const Ipp64s* pSrcHigh,
     Ipp64s* pDst, int lenDst);
IppStatus ippsWTHaarInv_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
     Ipp32f* pDst, int lenDst);
IppStatus ippsWTHaarInv_64f(const Ipp64f* pSrcLow, const Ipp64f* pSrcHigh,
     Ipp64f* pDst, int lenDst);
IppStatus ippsWTHaarInv_8s_Sfs(const Ipp8s* pSrcLow,
     const Ipp8s* pSrcHigh, Ipp8s* pDst, int lenDst, int scaleFactor);
IppStatus ippsWTHaarInv_16s_Sfs(const Ipp16s* pSrcLow,
     const Ipp16s* pSrcHigh, Ipp16s* pDst, int lenDst, int scaleFactor);
IppStatus ippsWTHaarInv_32s_Sfs(const Ipp32s* pSrcLow,
     const Ipp32s* pSrcHigh, Ipp32s* pDst, int lenDst, int scaleFactor);
IppStatus ippsWTHaarInv_64s_Sfs(const Ipp64s* pSrcLow,
     const Ipp64s* pSrcHigh, Ipp64s* pDst, int lenDst, int scaleFactor);
```

Arguments

pSrc	Pointer to the array which holds the input signal for the ippsWTHaarFwd function.
lenSrc	Number of elements in the source vector *pSrc*.
pDstLow	Pointer to the array which holds coarse “low frequency” components of the output of the ippsWTHaarFwd function.
pDstHigh	Pointer to the array which holds detail “high frequency” components of the output of the ippsWTHaarFwd function.
pSrcLow	Pointer to the array which holds coarse “low frequency” components of the input to the ippsWTHaarInv function.
pSrcHigh	Pointer to the array which holds detail “high frequency” components of the input to the ippsWTHaarInv function.
pDst	Pointer to the array which holds the output signal of the ippsWTHaarInv function.

lenDst	Number of elements in the destination vector *pDst*
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The ippsWTHaar family of functions perform forward and inverse single-level discrete Haar transforms. These transforms are orthogonal and reconstruct the original signal perfectly.

The forward transform can be considered as wavelet signal decomposition with lowpass decimation filter coefficients {1/2, 1/2} and highpass decimation filter coefficients {1/2, -1/2}.

The inverse transform is represented as wavelet signal reconstruction with lowpass interpolation filter coefficients {1, 1} and highpass interpolation filter coefficients {-1, 1}.

The decomposition filter coefficients are frequency response normalized to provide the same value range for both input and output signals. Thus, the amplitude of the low pass filter frequency response is 1 for zero-valued frequency, and the amplitude of the high pass filter frequency response is also 1 for the frequency value near to 0.5.

As the absolute values of the interpolation filter coefficients are equal to 1, the reconstruction of the signal requires few operations. It is well suited for usage in data compression applications. As the decomposition filter coefficients are powers of 2, the integer functions perform lossless decomposition with the *scaleFactor* value equal to -1. To avoid saturation, use higher-precision data types.

Note that the filter coefficients can be power spectral response normalized, see [Str96] for more information. Thus, the decomposition filter coefficients are $\{2^{-1/2}, 2^{-1/2}\}$ and $\{2^{-1/2}, -2^{-1/2}\}$; accordingly; the reconstruction filter coefficients are $\{2^{-1/2}, 2^{-1/2}\}$ and $\{-2^{-1/2}, 2^{-1/2}\}$.

In the following definition of the forward single-level discrete Haar transform, N=*lenSrc*. The coarse "low-frequency" component $c(k)$ is *pDstLow[k]* and the detail "high-frequency" component $d(k)$ is *pDstHigh[k]*; also $x(2k)$ and $x(2k+1)$ are even and odd values of the input signal *pSrc*, respectively.

$c(k)=(x(2k)+x(2k+1))/2$

$d(k)=(x(2k+1)-x(2k))/2$

In the inverse direction, $N=lenDst$. The coarse "low-frequency" component $c(k)$ is $pSrcLow[k]$ and the detail "high-frequency" component $d(k)$ is $pSrcHigh[k]$; also $y(2i)$ and $y(2i+1)$ are even and odd values of the output signal $pDst$, respectively.

$y(2i)=c(i)-d(i)$

$y(2i+1)=c(i)+d(i)$

For even length N, $0 \le k < N/2$ and $0 \le i < N/2$. Also, "low-frequency" and "high-frequency" components are of size $N/2$ for both original and reconstructed signals. The total length of components is equal to the signal length N.

In case of odd length N, the vector is considered as a vector of the extended length $N+1$ whose two last elements are equal to each other $x[N]=x[N-1]$. The last elements of the coarse and detail components of the decomposed signal are defined as follows:

$c((N+1)/2-1)=x(N-1)$

$d((N+1)/2-1)=0$

Correspondingly, the last element of the reconstructed signal is defined as:

$y(N)=y(N-1)=c((N+1)/2-1)$

For odd length N, $0 \le k < (N-1)/2$ and $0 \le i < (N-1)/2$, assuming that $c((N+1)/2-1)=x(N-1)$ and $y(N-1)=c((N+1)/2-1)$. The "low-frequency" component is of size $(N+1)/2$. The "high-frequency" component is of size $(N-1)/2$, because the last element $d((N+1)/2-1)$ is always equal to 0. The total length of components is also N.

Such an approach applies continuation of boundaries for filters having the symmetry properties, see [Bri94].

When performing block mode transforms, take into consideration that for decomposition and reconstruction of even-length signals no extrapolations at the boundaries is used. In case of odd-length signals, a symmetric continuation of the signal boundary with the last point replica is applied.

When it is necessary to have a continuous set of output blocks, all the input blocks are to be of even length, besides the last one (which can be either of odd or even length). Thus, if the whole amount of elements is odd, only the last block can be of odd length.

ippsWTHaarFwd. The function ippsWTHaarFwd performs the forward single-level discrete Haar transform of a *lenSrc* -length signal *pSrc* and stores the decomposed coarse “low-frequency” components in *pDstLow*, and the detail “high-frequency” components in *pDstHigh*.

ippsWTHaarInv. The function ippsWTHaarInv performs the inverse single-level discrete Haar transform of the coarse “low-frequency” components *pSrcLow* and detail “high-frequency” components *pSrcHigh*, and stores the reconstructed signal in the *lenDst* -length vector *pDst*.

Example 7-9 illustrates the use of the function ippsWTHaarFwd_32f.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than 4 for the function ippsWinBlackmanOpt and less than 3 for all other functions of the family.

Example 7-9 Using the ippsWTHaarFwd Function

```
IppStatus wthaar(void) {
      Ipp32f x[8], lo[4], hi[4];
      IppStatus status;
      ippsSet_32f(7, x, 8);  --x[4];
      status = ippsWTHaarFwd_32f(x, 8, lo, hi);
      printf_32f("WT Haar low  =", lo, 4, status);
      printf_32f("WT Haar high =", hi, 4, status);
      return status;
}
Output:
    WT Haar low  =  7.000000 7.000000 6.500000 7.000000
    WT Haar high =  0.000000 0.000000 0.500000 0.000000
```

Related Topics

For more information on wavelet transforms, see: [Str96], pp. 153-157, *Wavelet and Filter Banks*, Wellesley-Cambridge Press; and [Bri94], *Classification of Nonexpansive Symmetric Extension Transforms for Multirate Filter Banks*, Los Alamos Report LA-UR-94-1747, 1994. For more information on these references, see the Bibliography at the end of this manual.

Transforms for User Filter Banks

This section describes the functions that perform forward or inverse wavelet transforms for user filter banks.

WTFwdInitAlloc, WTInvInitAlloc

Initializes the wavelet transform state.

```
IppStatus ippsWTFwdInitAlloc_32f(IppsWTFwdState_32f** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_8s32f(IppsWTFwdState_8s32f** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_8u32f(IppsWTFwdState_8u32f** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_16s32f(IppsWTFwdState_16s32f** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_16u32f(IppsWTFwdState_16u32f** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);


IppStatus ippsWTInvInitAlloc_32f(IppsWTInvState_32f** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTInvInitAlloc_32f8s(IppsWTInvState_32f8s** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTInvInitAlloc_32f8u(IppsWTInvState_32f8u** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTInvInitAlloc_32f16s(IppsWTInvState_32f16s** pState,
     const Ipp32f* pTapsLow, int lenLow, int offsLow,
     const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTInvInitAlloc_32f16u(IppsWTInvState_32f16u** pState,
     const Ipp32f* pTapsLow, int lenLow, int offsLow,
     const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

Arguments

pState	Pointer to the location to store the allocated and initialized state structure.
pTapsLow	Pointer to the vector of lowpass filter taps.
lenLow	Number of taps in the lowpass filter.
offsLow	Additional delay (offset) of the lowpass filter.
pTapsHigh	Pointer to the vector of highpass filter taps.
lenHigh	Number of taps in the highpass filter.
offsHigh	Additional delay (offset) of the highpass filter.

Discussion

The functions `ippsWTFwdInitAlloc` and `ippsWTInvInitAlloc` create and initialize the WT state structure *pState* with the following parameters: the lowpass and highpass filter taps *pTapsLow* and *pTapsHigh*, lengths *lenLow* and *lenHigh*, input additional delays *offsLow* and *offsHigh*, respectively.

ippsWTFwdInitAlloc. The function `ippsWTFwdInitAlloc` initializes the forward WT state structure.

ippsWTInvInitAlloc. The function `ippsWTInvInitAlloc` initializes the inverse WT state structure.

Application Notes

These functions initialize the wavelet transform state structure, allocate memory for the state structure, initialize it, and returns the *pState* pointer to the state structure. The initialization procedures are implemented separately for forward and inverse transforms. To perform both forward and inverse wavelet transforms, create two

separate state structures. In general, the meanings of initialization parameters of forward and inverse transforms are similar. Each function has parameters describing of a pair of filters. The forward transform uses the taps *pTapsHigh* and *pTapsLow*, and the lengths *lenHigh* and *lenLow* of a pair of analysis filters. The inverse transform uses the taps *pTapsHigh* and *pTapsLow*, and the lengths *lenHigh* and *lenLow* of a pair of synthesis filters. Besides lengths and sets of taps the functions allow to specify an additional delay *offsLow* and *offsHigh* for each filter. The adjustable values of delays allow to synchronize:

- group delays for highpass and lowpass filters;
- delays between data of different levels in multilevel decomposition and reconstruction algorithms.

For more information about using these parameters, see "WTFwd" on page 7-63 and "WTInv" on page 7-70. The minimum allowed value of the additional delay for the forward transform is -1. For the inverse transform the delay values must be greater or equal to 0. See "WTFwd" on page 7-63 and "WTInv" on page 7-70 for an example showing how to choose additional delay values. The initialization functions copy filter taps into the state structure *pState*. So all the memory referred to with the pointers can be freed or modified after the functions finished operating. In case of the memory shortage, the function sets a zero pointer to the structure.

Boundaries extrapolation. Typically, reversible wavelet transforms of a bounded signal require data extrapolation towards one or both sides. All internal delay lines are set to zero at the stage of initialization. To set a non-zero signal prehistory, call the function `ippsWTFwdSetDlyLine` on page 7-67. When processed an entire limited data set, data extrapolation may be performed both towards the start and the end of the data vector. For that, the source data and their initial extrapolation are used to form the delay line, the rest of the signal is subdivided into the main block and the signal end. The signal end data and their extrapolation are used to form the last block.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState*, *pTapsHigh*, or *pTapsLow* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *lenLow* or *lenHigh* is less than or equal to 0.

ippStsWtOffsetErr	Indicates an error when the filter delay *offsLow* or *offsHigh* is less than -1 for the forward transform; and is less than 0 for the inverse transform.

WTFwdFree, WTInvFree

Closes a wavelet transform state.

```
IppStatus ippsWTFwdFree_32f(IppsWTFwdState_32f* pState);
IppStatus ippsWTFwdFree_8s32f(IppsWTFwdState_8s32f* pState);
IppStatus ippsWTFwdFree_8u32f(IppsWTFwdState_8u32f* pState);
IppStatus ippsWTFwdFree_16s32f(IppsWTFwdState_16s32f* pState);
IppStatus ippsWTFwdFree_16u32f(IppsWTFwdState_16u32f* pState);
IppStatus ippsWTInvFree_32f(IppsWTInvState_32f* pState);
IppStatus ippsWTInvFree_32f8s(IppsWTInvState_32f8s* pState);
IppStatus ippsWTInvFree_32f8u(IppsWTInvState_32f8u* pState);
IppStatus ippsWTInvFree_32f16s (IppsWTInvState_32f16s* pState);
IppStatus ippsWTInvFree_32f16u(IppsWTInvState_32f16u* pState);
```

Arguments

pState	Pointer to the state structure to be closed.

Discussion

The function ippsWTFwdFree and ippsWTInvFree close the WT state structure *pState* by freeing all the internal memory associated with the state created by ippsWTFwdInitAlloc or ippsWTInvInitAlloc. Call either ippsWTFwdFree or ippsWTInvFree after the transform is completed. If the *pState* pointer is NULL, the function performs no operation and returns the ippStsNullPtrErr status.

ippsWTFwdFree. The function ippsWTFwdFree closes the forward WT state structure.

ippsWTInvFree. The function ippsWTInvFree closes the inverse WT state structure.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pState* is NULL.
ippStsStateMatchErr	Indicates an error when the state identifier *pState* is incorrect.

WTFwd

Computes the forward wavelet transform.

```
IppStatus ippsWTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDstLow,
Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_32f* pState);
IppStatus ippsWTFwd_8s32f(const Ipp8s* pSrc, Ipp32f* pDstLow,
Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_8s32f* pState);
IppStatus ippsWTFwd_8u32f(const Ipp8u* pSrc, Ipp32f* pDstLow,
Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_8u32f* pState);
IppStatus ippsWTFwd_16s32f(const Ipp16s* pSrc, Ipp32f* pDstLow,
Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_16s32f* pState);
IppStatus ippsWTFwd_16u32f(const Ipp16u* pSrc, Ipp32f* pDstLow,
Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_16u32f* pState);
```

Arguments

pSrc	Pointer to the vector which holds the input signal for decomposition.
pDstLow	Pointer to the vector which holds output coarse “low frequency” components.
pDstHigh	Pointer to the vector which holds output detail “high frequency” components.
dstLen	Number of elements in the vectors *pDstHigh* and *pDstLow*.
pState	Pointer to the state structure.

Discussion

The function ippsWTFwd computes the forward wavelet transform. The function transforms the (2 * *dstLen*) -length source data block *pSrc* into “low frequency” components *pDstLow* and “high frequency” components *pDstLow*. The transform parameters are specified in the state structure *pState*.

Application Notes

These functions perform the one-level forward discrete multi-scale transform. An equivalent transform diagram is shown in Figure 7-4. The input signal is divided into the “low frequency” and “high frequency” components. The transfer characteristics of filters are defined by the coefficients set at the initialization stage. The functions are designed for the block processing of data; the transform state structure *pState* contains all needed filter delay lines. Besides these main delay lines each function has an additional delay line for each filter. Adjustable extra delay lines help synchronize group delay times of both highpass and lowpass filters. Moreover, in multilevel systems of signal decomposition delays between different decomposition levels may also be synchronized.

Input and output data block lengths. The functions are designed to decompose signal blocks of even length, therefore, these functions have one parameter only, i.e. the length of input components. The length of the input block must be double the size of each component.

Filter group delays synchronization. Some applications may require synchronization of highpass and lowpass filter time responses. A typical example of this synchronization is synchronizing symmetrical filters of different length.

Below follows an example of biorthogonal set of spline filters of respective length of 6 and 2:

```
static const float decLow[6] =
{
    -6.25000000e-002f,
     6.25000000e-002f,
     5.00000000e-001f,
     5.00000000e-001f,
     6.25000000e-002f,
     -6.25000000e-002f
};


static const float decHigh[2] =
{
    -5.00000000e-001f,
     5.00000000e-001f
};
```

In this case the lowpass filter gives a delay two samples longer than the highpass filter, which is exactly what the difference between additional initialization function delays should be. The following values must be selected to ensure minimum common signal delay, *offsLow* = -1, *offsHigh* = -1 + 2 = 1. In this case the group times of filter delays are balanced by additional delays. The total delay time is equal to the lowpass filter group delay which has the value of two samples in the decomposition stage in the original signal time frame.

NOTE. *Biorthogonal and orthogonal filter banks are distinguished by one specific peculiarity, that is, forward transform additional delays must be uniformly even for faultless signal reconstruction.*

Multilevel decomposition algorithm. The implementation of multilevel decomposition algorithms may require synchronization of signal delays across components of different levels.

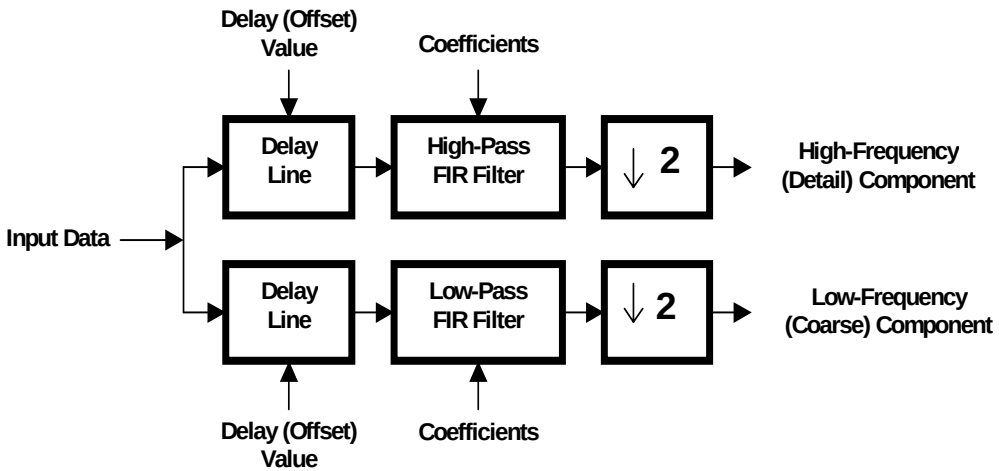
This is illustrated in the example of the three-level decomposition shown in Figure 7-2. Assume that for transformation the biorthogonal set of spline filters with respective filter length of 6 and 2 is used. Since group delay definitely needs to be synchronized, for the last level select additional filter delays *offsLow3* = -1, *offsHigh3* = 1. Total delay at the last stage of decomposition for this set of filters is two samples. This value corresponds to the time scale of the input of the last stage of decomposition. In order to ensure an equivalent delay of the "detail" part on the second level, the delay must be increased by 2*2 samples. Respective values of additional delays for the second level is equal to *offsLow2* = -1, *offsHigh2* = *offsHigh3* + 4 = 5. A greater value of the "high frequency" component delay needs to be selected for the first level of decomposition, *offsLow1* = -1, *offsHigh1* = *offsHigh2* + 2*4 = 13.

Total delay for three levels of decomposition is equal to 12 samples.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data blocks are `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier *pState* is incorrect.
`ippStsSizeErr`	Indicates an error when *dstLen* or *srcLen* is less than or equal to 0.

Figure 7-4 One Level Forward Wavelet Transform



WTFwdSetDlyLine, WTFwdGetDlyLine

Sets and gets the delay lines of the forward wavelet transform.

```
IppStatus ippsWTFwdSetDlyLine_32f(IppsWTFwdState_32f* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_8s32f(IppsWTFwdState_8s32f* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_8u32f(IppsWTFwdState_8u32f* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_16s32f(IppsWTFwdState_16s32f* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
```

```
IppStatus ippsWTFwdSetDlyLine_16u32f(IppsWTFwdState_16u32f* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);


IppStatus ippsWTFwdGetDlyLine_32f(IppsWTFwdState_32f* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_8s32f(IppsWTFwdState_8s32f* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_8u32f(IppsWTFwdState_8u32f* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_16s32f(IppsWTFwdState_16s32f* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_16u32f(IppsWTFwdState_16u32f* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
```

Arguments

pState	Pointer to the state structure.
pDlyLow	Pointer to the vector which holds the delay lines for “low frequency” components.
pDlyHigh	Pointer to the vector which holds the delay lines for “high frequency” components.

Discussion

The functions ippsWTFwdSetDlyLine and ippsWTFwdSetDlyLine copy the delay line values from *pDlyHigh* and *pDlyLow*, and stores them into the state structure *pState*.

ippsWTFwdSetDlyLine. The functions ippsWTFwdSetDlyLine sets the delay line values of the forward WT state.

ippsWTFwdGetDlyLine. The functions ippsWTFwdSetDlyLine sets the delay line values of the inverse WT state.

Application Notes

These functions are designed to shape the signal prehistory, save and reconstruct delay lines. Delay lines are implemented separately for highpass and lowpass filters, which gives the option of getting independent signal prehistories for each filter.

Delay line data format. Despite that any delay line formats could be used inside transformations, the functions provide the simplest format of received and returned vectors. Data either transferred to or returned from the delay lines have the same format as the initial signal fed into the forward transform functions, i.e., delay line vectors must be made up of a succession of the signal prehistory counts in the same time frame as the initial signal.

Delay line lengths. The length of the vectors that are transferred to or received by the delay line installation or reading functions is uniquely defined by the filter length and the value of additional filter delay.

The following expression defines the length of the delay line vector of the “low frequency” component filter:

dlyLowLen = *lenLow* + *offsLow* - 1,

where *lenLow* and *offsLow* are respectively the length and additional delay of the “low frequency” component filter.

The following expression defines the length of the delay line vector of the “high frequency” component filter:

dlyHighLen = *lenHigh* + *offsHigh* - 1,

where *lenHigh* and *offsHigh* are respectively the length and additional delay of the “high frequency” component filter.

The *lenLow*, *offsLow*, *lenHigh*, and *offsHigh* parameters are also described in the section on the transform initialization, see “WTFwdInitAlloc, WTInvInitAlloc” in page 7-59.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pDlyLow* or *pDlyHigh* is `NULL`.
`ippStsStateMatchErr`	Indicates an error when the state identifier *pState* is incorrect.

WTInv

Computes the inverse wavelet transform.

```
IppStatus ippsWTInv_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
     int srcLen, Ipp32f* pDst, IppsWTInvState_32f* pState);
IppStatus ippsWTInv_32f8s(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
     int srcLen, Ipp8s* pDst, IppsWTInvState_32f8s* pState);
IppStatus ippsWTInv_32f8u(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
     int srcLen, Ipp8u* pDst, IppsWTInvState_32f8u* pState);
IppStatus ippsWTInv_32f16s(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
     int srcLen, Ipp16s* pDst, IppsWTInvState_32f16s* pState);
IppStatus ippsWTInv_32f16u(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
     int srcLen, Ipp16u* pDst, IppsWTInvState_32f16u* pState);
```

Arguments

pSrcLow	Pointer to the vector which holds input coarse “low frequency” components.
pSrcHigh	Pointer to the vector which holds detail “high frequency” components.
srcLen	Number of elements in the vectors *pSrcHigh* and *pSrcLow*.
pDst	Pointer to the vector which holds the output reconstructed signal.
pState	Pointer to the state structure.

Discussion

The function ippsDCTInv computes the inverse wavelet transform. The function transforms the “low frequency” components *pSrcLow* and “high frequency” components *pSrcHigh* into the (2 * *srcLen*)-length destination data block *pDst*. The transform parameters are specified in the state structure *pState*.

Application Notes

These functions are used for one level of inverse multiscale transformation which results in reconstructing the original signal from the two “low frequency” and “high frequency” components. Figure 7-5 shows an equivalent transform algorithm. Two interpolation filters are used for signal reconstruction; their coefficients are set at the initialization stage. The inverse transform implementation, similar to forward transform implementation, contains additional delay lines needed to synchronize the group time of filter delays and delays across different levels of data reconstruction.

Input and output data block lengths. These functions are designed to reconstruct the blocks of the even length signal. The signal component length must be the input data. The length of the output block of the reconstructed signal must be double the length of each of the components.

Filter group delay synchronization. In this example consider a biorthogonal set of spline filters of length 2 and 6:

```
static const float recLow[2] =
{
     1.00000000e+000f,
     1.00000000e+000f
};
static const float recHigh[6] =
{
    -1.25000000e-001f,
    -1.25000000e-001f,
     1.00000000e+000f,
    -1.00000000e+000f,
     1.25000000e-001f,
     1.25000000e-001f
};
```

This set of filters corresponds to the set of filters considered in a similar section of the forward transform, see “WTFwd” on page 7-63.

Unlike the case described above, this time the highpass filter generates a delay greater by two samples compared against the low frequency filter. The two sample difference should also exist between initialization function additional delays. The following parameters of additional delays need to be selected in order to ensure the minimum

total delay, *offsLow* = 2, *offsHigh* = 0. In this case the total delay is equal to the highpass filter group delay, which at the decomposition stage is equal to two samples in the original signal time frame.

Total delay of one level of decomposition and reconstruction is equal to 4 samples, considering the decomposition stage delay.

NOTE. *Biorthogonal and orthogonal filter banks are distinguished by one specific peculiarity, that is, inverse transform additional delays must be uniformly even and opposite to the evenness of the decomposition delays for faultless signal reconstruction.*

Multilevel reconstruction algorithms. An example of a three-level signal reconstruction algorithm is shown in Figure 7-3. The scheme corresponds to the decomposition scheme described in the section on the forward transform, see "WTFwd" on page 7-63. Therefore, for the inverse transform the biorthogonal set of spline filters with respective filter length of 6 and 2 is used. The lowest level filter delays are set to *offsLow3* = 2, *offsHigh3* = 0. The total delay at this stage of reconstruction is equal to two samples. In order to ensure an equivalent delay of the "detail" part in the middle level, the delay must be increased. Respective values of additional delays for the second level are equal to *offsLow2* = 2, *offsHigh2* = *offsHigh3* + 2*2 = 4. A greater value of high frequency component delay needs to be selected for the last level of reconstruction, *offsLow1* = -1, *offsHigh1* = *offsHigh2* + 2*4 = 12.

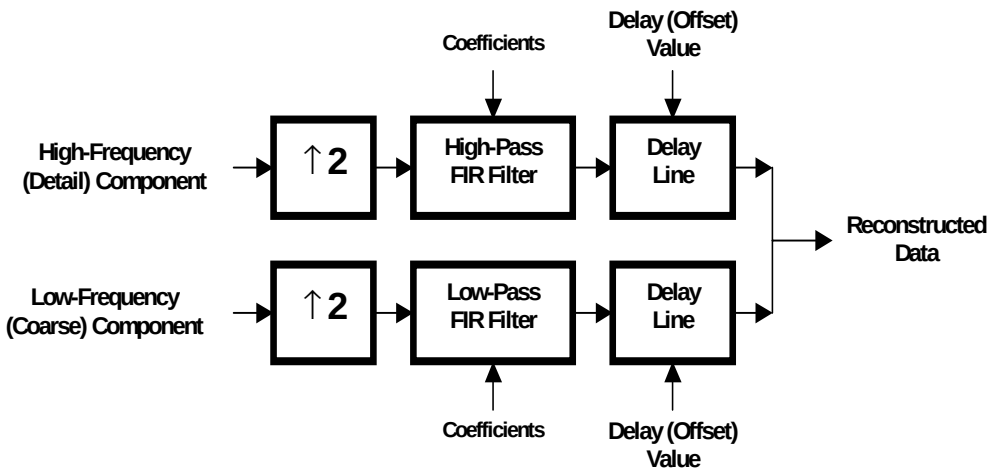
The total delay for three levels of reconstruction is equal to 12 samples. The total delay of the three-level decomposition and reconstruction cycle is equal to 24 samples.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data blocks are `NULL`.
`ippStsStateMatchErr`	Indicates an error when the state identifier *pState* is incorrect.

`ippStsSizeErr`	Indicates an error when *`dstLen`* or *`srcLen`* is less than or equal to 0.

Figure 7-5 One Level Inverse Wavelet Transform



WTInvSetDlyLine, WTInvGetDlyLine

Sets and gets the delay lines of the inverse wavelet transform.

```
IppStatus ippsWTInvSetDlyLine_32f(IppsWTInvState_32f* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f8s(IppsWTInvState_32f8s* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f8u(IppsWTInvState_32f8u* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
```

```
IppStatus ippsWTInvSetDlyLine_32f16s(IppsWTInvState_32f16s* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f16u(IppsWTInvState_32f16u* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippsWTInvGetDlyLine_32f(IppsWTInvState_32f* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f8s(IppsWTInvState_32f8s* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f8u(IppsWTInvState_32f8u* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f16s(IppsWTInvState_32f16s* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f16u(IppsWTInvState_32f16u* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
```

Arguments

pState	Pointer to the state structure.
pDlyLow	Pointer to the vector which holds delay lines for “low frequency” components.
pDlyHigh	Pointer to the vector which holds delay lines for “high frequency” components.

Discussion

The functions ippsWTFwdSetDlyLine and ippsWTFwdSetDlyLine copy the delay line values from *pDlyHigh* and *pDlyLow*, and stores them into the state structure *pState*.

ippsWTFwdSetDlyLine. The functions ippsWTFwdSetDlyLine sets the delay line values of the forward WT state.

ippsWTInvSetDlyLine. The functions ippsWTFwdSetDlyLine sets the delay line values of the inverse WT state.

Application Notes

These functions set and read delay lines of inverse multiscale transformation. The functions receive or return filter low and high frequency component delay line vectors. The functions may be used to shape previous history of each of the components. Installation functions and read functions together ensure that delay lines from each filter are saved and reconstructed.

Delay line data format. Despite that any delay line formats could be used inside transformations, the functions provide the simplest format of received and returned vectors. Data either transferred to or returned from the delay lines have the same format as the low and high frequency components at the input of the inverse transform functions. Thus, delay line vectors must be made up of a succession of signal prehistory counts in the same time frame as the input components.

Delay line lengths. The length of the vectors that are transferred to or received by the delay line installation or reading functions is uniquely defined by the filter length and the value of additional filter delay.

The following expression defines the length of the delay line vector of the “low frequency” component filter in terms of the C language (integer division by two is used here for simplicity):

dlyLowLen = (*lenLow* + *offsLow* - 1) / 2,

where *lenLow* and *offsLow* are respectively the length and additional delay of the “low frequency” component filter.

The following expression defines the length of the delay line vector of the “high frequency” component filter in terms of the C language:

dlyHighLen = (*lenHigh* + *offsHigh* - 1) / 2,

where *lenHigh* and *offsHigh* are respectively the length and additional delay of the “high frequency” component filter.

The *lenLow*, *offsLow*, *lenHigh*, and *offsHigh* parameters are also described in the section on the transform initialization, see “WTFwdInitAlloc, WTInvInitAlloc” on page 7-59.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pDlyLow`* or *`pDlyHigh`* is `NULL`.
`ippStsStateMatchErr`	Indicates an error when the state identifier *`pState`* is incorrect.

Speech Recognition Functions

8

This chapter describes IPP functions that are commonly used in speech recognition applications.

NOTE. *In the below descriptions of function arguments, when an argument refers to a vector or an array, the expression in square brackets given after the explanation of the argument specifies the dimension of that vector or array.*
In this chapter, unlike other chapters of this manual, arguments that represent step values are measured in elements of the corresponding array, unless explicitly stated to the contrary.

Basic Arithmetics

The functions included in this section perform generic arithmetic operations on vectors and matrices.

AddAllRowSum

Calculates the sums of column vectors in a matrix and adds the sums to a vector.

```
IppStatus ippsAddAllRowSum_32f_D2(const Ipp32f* pSrc, int step,
   int height, Ipp32f* pSrcDst, int width);
IppStatus ippsAddAllRowSum_32f_D2L(const Ipp32f** mSrc, int height,
   Ipp32f* pSrcDst, int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
step	Row step in *pSrc*.
height	Number of rows in the input matrix *mSrc*.
pSrcDst	Pointer to the output vector [*width*].
width	Number of columns in the input matrix *mSrc*, and also the length of the output vector *pSrcDst*.

Discussion

The function ippsAddAllRowSum calculates sums of column vectors in the input matrix, and adds these sums to the output vector. The operations are as follows:

For functions with the D2 suffix,

$$pSrcDst[j] = pSrcDst[j] + \sum_{i=0}^{height-1} pSrc[i \cdot step + j],\ 0 \le j < width\ .$$

For functions with the D2L suffix,

$$pSrcDst[j] = pSrcDst[j] + \sum_{i=0}^{height-1} mSrc[i][j],\ 0 \le j < width\ .$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, or *pSrcDst* pointer is null.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

SumColumn

Calculates sums of column vectors in a matrix.

```
IppStatus ippsSumColumn_16s32s_D2Sfs(const Ipp16s* pSrc, int step, int height,
   Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsSumColumn_16s32f_D2(const Ipp16s* pSrc, int step, int height,
   Ipp32f* pDst, int width);
IppStatus ippsSumColumn_32f_D2(const Ipp32f* pSrc, int step, int height,
   Ipp32f* pDst, int width);
IppStatus ippsSumColumn_64f_D2(const Ipp64f* pSrc, int step, int height,
   Ipp64f* pDst, int width);
IppStatus ippsSumColumn_16s32s_D2LSfs(const Ipp16s** mSrc, int height,
   Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsSumColumn_16s32f_D2L(const Ipp16s** mSrc, int height,
   Ipp32f* pDst, int width);
IppStatus ippsSumColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f* pDst,
   int width);
IppStatus ippsSumColumn_64f_D2L(const Ipp64f** mSrc, int height, Ipp64f* pDst,
   int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
step	Row step in the input vector *pSrc*.
height	Number of rows in the input matrix *mSrc*.
pDst	Pointer to the output vector [*width*].
width	Number of columns in the input matrix *mSrc* and the length of the output vector *pDst*.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsSumColumn calculates sums of column vectors in the input matrix, and stores these sums in the output vector. The operations are as follows:

For functions with the D2 suffix,

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i \cdot step + j],\ 0 \le j < width\ .$$

For functions with the D2L suffix,

$$pDst[j] = \sum_{i=0}^{height-1} mSrc[i][j],\ 0 \le j < width\ .$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

SumRow

Calculates sums of row vectors in a matrix.

```
IppStatus ippsSumRow_16s32s_D2Sfs(const Ipp16s * pSrc, int height, int step,
   Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsSumRow_16s32f_D2(const Ipp16s* pSrc, int height, int step,
   Ipp32f* pDst, int width);
IppStatus ippsSumRow_32f_D2(const Ipp32f* pSrc, int height, int step,
   Ipp32f* pDst, int width);
IppStatus ippsSumRow_64f_D2(const Ipp64f* pSrc, int height, int step,
   Ipp64f* pDst, int width);
```

```
IppStatus ippsSumRow_16s32s_D2LSfs(const Ipp16s** mSrc, int height, Ipp32s*
   pDst, int width, int scaleFactor);
IppStatus ippsSumRow_16s32f_D2L(const Ipp16s** mSrc, int height,  Ipp32f* pDst,
   int width);
IppStatus ippsSumRow_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f* pDst,
   int width);
IppStatus ippsSumRow_64f_D2L(const Ipp64f** mSrc, int height, Ipp64f* pDst,
   int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
height	Number of rows in the input matrix *mSrc*, and also the length of the output vector *pDst*.
step	Row step in the input vector *pSrc*.
pDst	Pointer to the output vector [*height*].
width	Number of columns in the input matrix *mSrc*.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsSumRow calculates sums of row vectors in the input matrix *mSrc* and stores these sums in the output vector *pDst*. The operations are as follows:

For functions with the D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[i \cdot step + j], \quad 0 \le i < height .$$

For functions with the D2L suffix,

$$pDst[i] = \sum_{j=0}^{width-1} mSrc[i][j], \quad 0 \le i < height .$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

SubRow

Subtracts a vector from all matrix rows .

```
IppStatus ippsSubRow_16s_D2(const Ipp16s* pSrc, int width, Ipp16s* pSrcDst,
   int dstStep, int height);
IppStatus ippsSubRow_32f_D2(const Ipp32f* pSrc, int width, Ipp32f* pSrcDst,
   int dstStep, int height);
IppStatus ippsSubRow_16s_D2L(const Ipp16s* pSrc, Ipp16s** mSrcDst, int width,
   int height);
IppStatus ippsSubRow_32f_D2L(const Ipp32f* pSrc, Ipp32f** mSrcDst, int width,
   int height);
```

Arguments

pSrc	Pointer to the input vector [*width*].
pSrcDst	Pointer to the source and destination vector [*height*dstStep*].
mSrcDst	Pointer to the source and destination matrix [*height*][*width*].
width	Number of columns in the matrix *mSrcDst*.
dstStep	Row step in the vector *pSrcDst*.
height	Number of rows in the matrix *mSrcDst*.

Discussion

The function ippsSubRow subtracts the input vector *pSrc* from all matrix rows. The operations are as follows:

For functions with the D2 suffix,

$$pSrcDst[i \cdot dstStep + j] = pSrcDst[i \cdot dstStep + j] - pSrc[j],$$

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

For functions with the D2L suffix,

$$mSrcDst[i][j] = mSrcDst[i][j] - pSrc[j],$$

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

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pSrcDst*, or *mSrcDst* pointer is null.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *dstStep* is less than *width*.

CopyColumn_Indirect

Copies the input matrix with columns redirection.

```
IppStatus ippsCopyColumn_Indirect_16s_D2(const Ipp16s* pSrc, int srcLen,
   int srcStep, Ipp16s* pDst, const Ipp32s* pIndx, int dstLen, int dstStep, int
   height);
IppStatus ippsCopyColumn_Indirect_32f_D2(const Ipp32f* pSrc, int srcLen,
   int srcStep, Ipp32f* pDst, const Ipp32s* pIndx, int dstLen, int dstStep, int
   height);
IppStatus ippsCopyColumn_Indirect_16s_D2L(const Ipp16s** mSrc, int srcLen,
   Ipp16s** mDst, const Ipp32s* pIndx, int dstLen, int height);
```

```
IppStatus ippsCopyColumn_Indirect_32f_D2L(const Ipp32f** mSrc, int srcLen,
   Ipp32f** mDst, const Ipp32s* pIndx, int dstLen, int height);
```

Arguments

pSrc	Pointer to the input vector [*height***srcStep*].
mSrc	Pointer to the input matrix [*height*][*srcLen*].
srcLen	Number of columns in the input matrix *mSrc*.
srcStep	Row step in the vector *pSrc*.
pDst	Pointer to the output vector [*height***dstStep*].
mDst	Pointer to the output matrix [*height*][*dstLen*].
pIndx	Pointer to the redirection vector [*dstLen*].
dstLen	Number of columns in the output matrix *mDst*.
dstStep	Row step in the vector *pDst*.
height	Number of rows in both the input and output matrices.

Discussion

The function `ippsCopyColumn_Indirect` copies the input matrix *mSrc* to the output matrix *mDst* with the columns redirected by *pIndx*. The operations are as follows:

For functions with the D2 suffix,

$pDst[i \cdot dstStep + j] = pSrc[i \cdot srcStep + pIndx[j]]$, $0 \le i < height$, $0 \le j < dstLen$.

For functions with the D2L suffix,

$mDst[i][j] = mSrc[i][pIndx[j]]$, $0 \le i < height$, $0 \le j < dstLen$.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *mSrc*, *pDst*, *mDst*, or *pIndx* pointer is null.
`ippStsSizeErr`	Indicates an error when *height*, *srcLen*, or *dstLen* is less than or equal to 0, or $pIndx[j] \ge srcLen$ or $pIndx[j] < 0$ for $0 \le j < dstLen$.

ippStsStrideErr	Indicates an error when *srcStep* is less than *srcLen* or *dstStep* is less than *dstLen*.

BlockDMatrixInitAlloc

Initializes the structure that represents a symmetric block diagonal matrix.

```
IppStatus ippsBlockDMatrixInitAlloc_16s(IppsBlockDMatrix_16s** pMatrix,
   const Ipp16s** mSrc, const int* bSize, int nBlocks);
IppStatus ippsBlockDMatrixInitAlloc_32f(IppsBlockDMatrix_32f** pMatrix,
   const Ipp32f** mSrc, const int* bSize, int nBlocks);
IppStatus ippsBlockDMatrixInitAlloc_64f(IppsBlockDMatrix_64f** pMatrix,
   const Ipp64f** mSrc, const int* bSize, int nBlocks);
```

Arguments

pMatrix	Pointer to the block diagonal matrix to be created.
mSrc	Pointer to the vector of pointers to matrix rows.
bSize	Pointer to vector of block sizes [*nBlocks*].
nBlocks	Number of blocks in the matrix.

Discussion

The function ippsBlocDMatrixInitAlloc creates the structure that contains a symmetric block diagonal matrix. Matrix elements outside the blocks are assumed to be zero. The block diagonal matrix A is defined as follows:

$A[K_l + i][K_l + j] = A[K_l + j][K_l + i] = mSrc[K_l + i]\,[j]$,

for $i, j = 0 \ldots bSize[l] - 1$ and $l = 0 \ldots nBlocks - 1$,

where

$$K_l = \sum_{k<l} bSize[k]$$

and *mSrc*[K_l + *i*] points to the non-zero part of the matrix *A* row.
The size of the matrix is equal to $K_{nBlocks}$ by $K_{nBlocks}$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pMatrix* or *mSrc* pointer is null.
ippStsSizeErr	Indicates an error when *bSize* or *nBlocks* is less than or equal to 0.
ippStsMemAllocErr	Indicates an error when no memory was allocated.

BlockDMatrixFree

Deallocates the block diagonal matrix structure.

```
IppStatus ippsBlockDMatrixFree_16s(IppsBlockDMatrix_16s* pMatrix);
IppStatus ippsBlockDMatrixFree_32f(IppsBlockDMatrix_32f* pMatrix);
IppStatus ippsBlockDMatrixFree_64f(IppsBlockDMatrix_64f* pMatrix);
```

Arguments

pMatrix	Pointer to the block diagonal matrix.

Discussion

The function ippsBlockDMatrixFree destroys the block diagonal matrix structure, and frees all memory associated with it.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pMatrix* pointer is null.

Feature Processing

This section describes functions that pre-process raw speech signals. Feature extraction is the first step in the recognition process. Speech features are a compressed form of the original speech signals. By extracting features, the problem dimension is reduced. To some extent, feature extraction normalizes speaker variations and environmental distortions.
This section also includes functions needed to support both silence/speech detection and also some well-known analysis techniques on speech signals.

ZeroMean

Subtracts the mean value from the input vector.

```
IppStatus ippsZeroMean_16s(Ipp16s* pSrcDst, int len);
```

Arguments

pSrcDst	Pointer to the source and destination vector [*len*].
len	The number of elements in the vector.

Discussion

The function ippsZeroMean calculates the mean value of the *pSrcDst* vector and subtracts it from the vector *pSrcDst*. The resulting values are saturated if they exceed the range [-32768..32767] .

The operations are as follows:

$$pSrcDst[i] = \max(-32768, \min(32767, pSrcDst[i] - \frac{1}{len}\sum_{j=0}^{len-1} pSrcDst[j]))$$

for $0 \le i < len$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcDst* pointer is null.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

CompensateOffset

Removes the DC offset of the input signals.

```
IppStatus ippsCompensateOffset_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
   Ipp16s* pSrcDst0, Ipp16s dst0, Ipp32f val);
IppStatus ippsCompensateOffset_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
   Ipp32f* pSrcDst0, Ipp32f dst0, Ipp32f val);
IppStatus ippsCompensateOffset_16s_I(Ipp16s* pSrcDst, int len, Ipp16s*
   pSrcDst0, Ipp16s dst0, Ipp32f val);
IppStatus ippsCompensateOffset_32f_I(Ipp32f* pSrcDst, int len, Ipp32f*
   pSrcDst0, Ipp32f dst0, Ipp32f val);
```

Arguments

pSrc	Pointer to the source vector [*len*].
pDst	Pointer to the destination vector [*len*].
pSrcDst	Pointer to the source and destination vector [*len*] for in-place operations.
pSrcDst0	Pointer to the previous source element.
len	Number of elements in the vector.
dst0	Previous destination element.
val	Constant for offset compensation.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsCompensateOffset removes the offset of the input signals. The destination vector is calculated as follows.

For the ippsCompensateOffset function:

pDst[0] = *pSrc*[0] - *pSrcDst*[0] + *val*·*dst0*

$pDst[i] = pSrc[i] - pSrc[i-1] + val \cdot pDst[i-1]$, $1 \le i \le len - 1$

$pSrcDst0[0] = pSrc[len - 1]$,

and for ippsCompensateOffset_I function:

$y = pSrcDst[0] - pSrcDst0[0] + val \cdot dst0$, $x = pSrcDst[0]$, $pSrcDst[0] = y$,

$y = pSrcDst[i] - pSrcDst[i-1] + val \cdot x$, $x = pSrcDst[i]$, $pSrcDst[i] = y$,

$1 \le i \le len - 1$,

$pSrcDst0[0] = x$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, *pSrcDst*, or *pSrcDst0* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

SignChangeRate

Counts the zero-cross rate for the input signal.

```
IppStatus ippsSignChangeRate_16s(const Ipp16s* pSrc, int len, Ipp32s* pRes);
IppStatus ippsSignChangeRate_32f(const Ipp32f* pSrc, int len, Ipp32f* pRes);
IppStatus ippsSignChangeRateCount0_16s(const Ipp16s* pSrc, int len, Ipp32s*
   pRes);
IppStatus ippsSignChangeRateCount0_32f(const Ipp32f* pSrc, int len, Ipp32f*
   pRes);
```

Arguments

pSrc	Pointer to the input signal [*len*].
len	Number of elements in the input signal *pSrc*.
pRes	Pointer to the result variable.

Discussion

The function ippsSignChangeRate counts the number of sign changes in the input signal. This function can be used to detect speech in continuous speech input.
The operations are as follows:
For the ippsSignChangeRate function,

$$pRes[0] = \sum_{i=1}^{len-1} \begin{cases} 1, & if\ \ pSrc[i] \cdot pSrc[i-1] < 0 \\ 0, & otherwise \end{cases}$$

For the ippsSignChangeRateCount0 function,

$$pRes[0] = \frac{1}{2} \sum_{i=1}^{len-1} |sign(pSrc[i]) - sign(pSrc[i-1])| \ , \ sign(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pRes* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

LinearPrediction

Performs linear prediction analysis on the input vector.

```
IppStatus ippsLinearPrediction_Auto_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
   Ipp16s* pDst, int lenDst, int scaleFactor);
IppStatus ippsLinearPrediction_Auto_32f(const Ipp32f* pSrc, int lenSrc,
   Ipp32f* pDst, int lenDst);
IppStatus ippsLinearPrediction_Cov_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
   Ipp16s* pDst, int lenDst, int scaleFactor);
```

```
IppStatus ippsLinearPrediction_Cov_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f*
   pDst, int lenDst);
```

Arguments

pSrc	Pointer to the input vector [*lenSrc*].
lenSrc	Length of the input vector *pSrc*.
pDst	Pointer to the output LPC coefficients vector [*lenDst*].
lenDst	Length of the output vector *pDst*.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLinearPrediction performs linear prediction analysis on the input signal.

For functions with the Auto suffix, the LPC coefficients are calculated by solving the following equations that represent the autocorrelation approach:

$$\sum_{i=0}^{lenDst-1} pDst[i] \cdot r[|i-k|] = r[k] \text{ , } k = 0 \ldots lenDst \text{ ,}$$

where $r[k] = \sum_{j=0}^{lenSrc-k-1} pSrc[j] \cdot pSrc[j+k]$.

For functions with the Cov suffix, the LPC coefficients are calculated by solving the following equations that represent the covariance approach:

$$\sum_{i=0}^{lenDst-1} pDst[i] \cdot c[i][k] = c[0][k] \text{ , } k = 0 \ldots lenDst \text{ ,}$$

where $c[i][k] = \sum_{j=0}^{lenSrc-k-1} pSrc[j] \cdot pSrc[j+k-i]$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *lenSrc* or *lenDst* is less than or equal to 0, or *lenDst* is greater or equal than *lenSrc*.
ippStsNoOperation	Indicates no solution to the LPC problem.

Durbin

Performs Durbin's recursion on an input vector of autocorrelations.

```
IppStatus ippsDurbin_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
   Ipp32f* pErr, int scaleFactor);
IppStatus ippsDurbin_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
   Ipp32f* pErr);
```

Arguments

pSrc	Pointer to the input vector [*len+1*].
pDst	Pointer to the output LPC coefficients vector [*len*].
len	Length of the input and output vectors.
pErr	Pointer to the resulting prediction error.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsDurbin performs the Durbin's recursion on the input autocorrelation vector and calculates the linear prediction coefficients and the prediction error as follows:

1) Initialization:

$m = len,\ R_j = pSrc[j],\ j = 0,...m,\ E^0 = R_0$

2) Iterations for $i = 1,...m$:

$$k_i = \left(R_i - \sum_{j=1}^{i-1} y_j^{i-1} \cdot R_{i-j} \right) / E^{i-1} , \quad E^i = (1 - k_i^2) \cdot E^{i-1} ,$$

$$y_i^i = k_i , \quad y_j^i = y_j^{i-1} - k_i \cdot y_{i-j}^{i-1} , \; j = 1,... \; i - 1$$

3) Result:

$$pDst[i-1] = y_i^m , \; i = 1,...m, \quad pErr[0] = E^m .$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *pDst*, or *pErr* pointer is null.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsNoOperation`	Indicates no solution to the LPC problem ($E^i \leq 0$).

LPToCepstrum

Calculates cepstrum coefficients from linear prediction coefficients.

```
IppStatus ippsLPToCepstrum_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
   int scaleFactor);
IppStatus ippsLPToCepstrum_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the linear prediction coefficients [*len*].
pDst	Pointer to the cepstrum coefficients [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLPToCepstrum calculates the cepstrum coefficients from the linear prediction coefficients according to the following formula:

$$pDst[k] = -\left(pSrc[k] + \frac{1}{k+1} \sum_{i=1}^{k} (k-i+1) \cdot pSrc[i-1] \cdot pDst[k-i] \right), k = 0 \ldots len-1 .$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

CepstrumToLP

Calculates linear prediction coefficients from cepstrum coefficients.

```
IppStatus ippsCepstrumToLP_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
   int scaleFactor);
IppStatus ippsCepstrumToLP_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the cepstrum coefficients [*len*].
pDst	Pointer to the linear prediction coefficients [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsCepstrumToLP calculates the linear prediction coefficients from the cepstrum coefficients according to the following formula:

$$pDst[k] = -\left(pSrc[k] + \frac{1}{k+1} \sum_{i=1}^{k} (k-i+1) \cdot pSrc[k-i] \cdot pDst[i-1] \right), k = 0 \ldots len-1 .$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

LPToReflection

Calculates the linear prediction reflection coefficients from the linear prediction coefficients.

```
IppStatus ippsLPToReflection_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
   int scaleFactor);
IppStatus ippsLPToReflection_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the linear prediction coefficients [*len*].
pDst	Pointer to the linear prediction reflection coefficients [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLPToReflection calculates the linear prediction reflection coefficients according to the following formulae:

1) Initialization:

$n = len$, $a_i^n = pSrc[i-1]$, $i = 1,...n$

2) Iterations for $i = m,...1$:

$$k_i = a_i^i, \quad a_j^{i-1} = \frac{a_j^i - a_i^i \cdot a_{i-j}^i}{1 - k_i^2}, \quad j = 1,...i\text{-}1$$

3) Result:

$pDst[i-1] = k_i$, $i = 1,...n$.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is null.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsNoOperation`	Indicates that reflection coefficients could not be calculated (that is, $\|k_i\| = 1$ for one of iterations).

ReflectionToLP

Calculates the linear prediction coefficients from the linear prediction reflection coefficients.

```
IppStatus ippsReflectionToLP_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
   len, int scaleFactor);
IppStatus ippsReflectionToLP_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the linear prediction reflection coefficients [*len*].
pDst	Pointer to the linear prediction coefficients [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function `ippsReflectionToLP` calculates the linear prediction reflection coefficients from the linear prediction coefficients according to the following formulae:

1) Initialization:

$n = len$, $k_i = pSrc[i-1]$, $i = 1,...n$

2) Iterations for $i = 1,...m$:

$a_i^i = k_i$, $a_j^i = a_j^{i-1} - k_i \cdot a_{i-j}^{i-1}$, $j = 1,...k\text{-}1$

3) Result:

$pDst[i-1] = a_i^n$, $i = 1,...n$.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is null.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

MelToLinear

Converts Mel-scaled values to linear scale values.

```
IppStatus ippsMelToLinear_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
   Ipp32f melMul, Ipp32f melDiv);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pDst	Pointer to the output vector [*len*].
len	Length of input and output vectors.
melMul	Multiply factor in the Mel-scale equation.

melDiv	Divide factor in the Mel-scale equation.

Discussion

The function ippsMelToLinear converts the Mel-frequency scale to the linear frequency scale:

$$pDst[i] = melDiv \cdot \left[\exp\left(\frac{pSrc[i]}{melMul}\right) - 1\right], \quad i = 0,\ldots,len\text{-}1 .$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0, or *melMul* or *melDiv* is equal to 0.

LinearToMel

Converts linear-scale values to Mel-scale values.

```
IppStatus ippsLinearToMel_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
   melMul, Ipp32f melDiv);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pDst	Pointer to the output vector [*len*].
len	Length of the input and output vectors.
melMul	Multiply factor in the Mel-scale equation.
melDiv	Divide factor in the Mel-scale equation.

Discussion

The function ippsLinearToMel converts the linear frequency scale to the Mel-frequency scale:

$pDst[i] = melMul \cdot \ln\left(1 + \frac{pSrc[i]}{melDiv}\right)$, $i = 0,\ldots,len\text{-}1$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0, or *melMul* or *melDiv* is equal to 0.

CopyWithPadding

Copies the input signal to the output with zero-padding.

```
IppStatus ippsCopyWithPadding_16s(const Ipp16s* pSrc, int lenSrc, Ipp16s*
   pDst, int lenDst);
IppStatus ippsCopyWithPadding_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f*
   pDst, int lenDst);
```

Arguments

pSrc	Pointer to the input vector [*lenSrc*].
pDst	Pointer to the output vector [*lenDst*].
lenSrc	Length of the input vector *pSrc*.
lenDst	Length of the output vector *pDst*.

Discussion

The function ippsCopyWithPadding copies the input vector to the output vector. If the length of the output vector is bigger, the trailing elements are padded with zeroes.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.

ippStsSizeErr	Indicates an error when *lenSrc* or *lenDst* is less than or equal to 0, or when *lenDst* is less than *lenSrc*.

MelFBankInitAlloc

Initializes the structure for performing the Mel frequency filter bank analysis.

```
IppStatus ippsMelFBankInitAlloc_16s(IppsFBankState_16s** pFBank,
   int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f
   highFreq, int nFilter, Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
IppStatus ippsMelFBankInitAlloc_32f(IppsFBankState_32f** pFBank,
   int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f
   highFreq, int nFilter, Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
```

Arguments

pFBank	Pointer to the Mel-scale filter bank structure to be created.
pFFTLen	Pointer to the length of FFT (*N* = *pFFTLen*[0]) used for the filter bank evaluation.
winSize	Frame length (in samples).
sampFreq	Input signal sampling frequency f_i (in Hz).
lowFreq	Start frequency f_{low} of the first band-pass filter (in Hz).
highFreq	End frequency f_{high} of the last band-pass filter (in Hz).
nFilter	Number of Mel-scale filter banks *K*.
melMul	Mel-scale formula multiply factor.
melDiv	Mel-scale formula divisor.
mode	Flags that determine the execution mode; can have the following values:
	IPP_FBANK_MELWGT – the function calculates filter bank weights in Mel-scale;
	IPP_FBANK_FREQWGT – the function calculates filter bank weights in the frequency space.
	One of the above two flags should necessarily be set.

IPP_POWER_SPECTRUM – indicates that the FFT power spectrum is used during the filter bank analysis.

Discussion

The function ippsMelFBankInitAlloc initializes the triangular filter banks for the Mel-frequency filter bank analysis. The filter bank analysis is one of the major steps in the Mel-Frequency Cepstrum Coefficients (MFCC) feature calculation.

The Mel-frequency translation is accomplished according to the following equation:

$$mel(f) = melMul \cdot \ln\left(1 + \frac{f}{melDiv}\right)$$

The center of each filter bank (c_k in Mel-scale and y_k in FFT domain) is calculated as follows:

$$c_k = mel(f_{low}) + \frac{k}{K+1} \cdot [mel(f_{high}) - mel(f_{low})] ,$$

$$y_k = round\left\{\frac{N}{f_s} mel^{-1}(c_k)\right\} ,$$

for $k = 0...K+1$,

where *round*(x) rounds the value of x to the nearest integer, and N is the length of FFT.

If *mode* is IPP_FBANK_FREQWGT, then the filter outputs will be calculated (by the function ippsEvalFBank) according to the following formula:

$$fFank_{k-1} = \sum_{i=y_{k-1}}^{y_k} \frac{i - y_{k-1} + 1}{y_k - y_{k-1} + 1} \cdot x_i + \sum_{i=y_k+1}^{y_{k+1}} \frac{y_{k+1} - i + 1}{y_{k+1} - y_k + 1} \cdot x_i , \text{ for } k = 1... K , \quad (8.1)$$

where x_i is the magnitude of the corresponding FFT spectrums.

If *mode* is IPP_FBANK_MELWGT, the filter outputs will be calculated (by the function ippsEvalFBank) according to the formula:

$$fFank_{k-1} = \sum_{i=y_{k-1}}^{y_k} \frac{mel(i) - c_{k-1}}{c_k - c_{k-1}} \cdot x_i + \sum_{i=y_k+1}^{y_{k+1}} \frac{c_{k+1} - mel(i)}{c_{k+1} - c_k} \cdot x_i , \text{ for } k = 1... K . \quad (8.2)$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pFFTLen* pointer is null.
ippStsSizeErr	Indicates an error when *winSize*, *nFilter*, *sampFreq*, or *lowFreq* is less than or equal to 0.
ippStsFBankFreqErr	Indicates an error when *highFreq* is less than *lowFreq* or *highFreq* is greater than *sampFreq/2*.
ippStsFBankFlagErr	Indicates an error when the *mode* value is incorrect.
ippStsMemAllocErr	Indicates an error when no memory was allocated.

MelLinFBankInitAlloc

Initializes the structure for performing a combined linear and Mel-frequency filter bank analysis.

```
IppStatus ippsMelLinFBankInitAlloc_16s(IppsFBankState_16s** pFBank,
   int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq,
   Ipp32f highFreq, int nFilter, Ipp32f highLinFreq, int nLinFilter,
   Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
IppStatus ippsMelLinFBankInitAlloc_32f(IppsFBankState_32f** pFBank,
   int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq,
   Ipp32f highFreq, int nFilter, Ipp32f highLinFreq, int nLinFilter,
   Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
```

Arguments

pFBank	Pointer to the filter bank structure to be created.
pFFTLen	Pointer to the length of FFT (N = *pFFTLen*[0]) used for the filter bank evaluation.
winSize	Frame length (in samples).
sampFreq	Input signal sampling frequency f_i (in Hz).

`lowFreq`	Start frequency f_{low} of the first band-pass filter (in Hz).
`highLinFreq`	End frequency f_{lin} of the last band-pass filter in linear scale (in Hz).
`highFreq`	End frequency f_{high} of the last band-pass filter (in Hz).
`nFilter`	Number of Mel-scale filter banks K.
`nLinFilter`	Number of linear-scale filter banks L.
`melMul`	Mel-scale formula multiply factor.
`melDiv`	Mel-scale formula divisor.
`mode`	Flags determining the function's execution mode; can have the following values:

`IPP_FBANK_MELWGT` – the function calculates filter bank weights in Mel-scale;

`IPP_FBANK_FREQWGT` – the function calculates filter bank weights in the frequency space.

One of the above two flags should necessarily be set.

`IPP_POWER_SPECTRUM` – indicates that the FFT power spectrum is used during the filter bank analysis.

Discussion

The function `ippsMelLinFBankInitAlloc` initializes the linear and Mel-frequency triangular filter banks. The first L filters are placed linearly on the frequency band from f_{low} to f_{lin}. The center of each filter bank (c_k in Mel-scale and y_k in FFT domain) is calculated as follows:

If $L = K$, then

$$z_k = f_{low} + \frac{f_{lin} - f_{low}}{L+1} \cdot k\ , \quad y_k = round\left\{\frac{N}{f_s} \cdot z_k\right\}\ , \quad k = 0,...L+1$$

If $L < K$, then

$$z_k = f_{low} + \frac{f_{lin} - f_{low}}{L} \cdot k\ , \quad y_k = round\left\{\frac{N}{f_s} \cdot z_k\right\}\ , \quad k = 0,...L$$

and

$$c_k = mel(f_{lin}) + \frac{k-L}{K-L+1} \cdot [mel(f_{high}) - mel(f_{lin})]\ ,$$

$$y_k = round\left\{\frac{N}{f_s} \cdot mel^{-1}(c_k)\right\}, \text{ for } k = L,...K+1 ,$$

where *round*(*x*) rounds the value of *x* to the nearest integer, and *N* is the FFT length.

If *mode* is `IPP_FBANK_FREQWGT`, the filter outputs are calculated (by the function `ippsEvalFBank`) according to the formula (8.1) given for the `ippsMelFBankInitAlloc` function.

If *mode* is `IPP_FBANK_MELWGT`, the filter output is calculated (by the function `ippsEvalFBank`) according to the following formulae:

If $L = K$, then

$$fFank_{k-1} = \sum_{i=y_{k-1}}^{y_k} \frac{i - z_{k-1}}{z_k - z_{k-1}} \cdot x_i + \sum_{i=y_k+1}^{y_{k+1}} \frac{z_{k+1} - i}{z_{k+1} - z_k} \cdot x_i , \quad k = 1,...K \qquad (8.3)$$

If $L < K$, then:

for filter outputs 1,...,*L*-1 the above formula (8.3) is used;

for $k = L$, the output is calculated as

$$fFank_{L-1} = \sum_{i=y_{L-1}}^{y_L} \frac{i - z_{L-1}}{z_L - z_{L-1}} \cdot x_i + \sum_{i=y_L+1}^{y_{L+1}} \frac{c_{L+1} - mel(i)}{c_{L+1} - c_L} \cdot x_i ;$$

for filter outputs *L*+1,...,*K* the formula (8.1) is used.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pFBank* or *pFFTLen* pointer is null.
`ippStsSizeErr`	Indicates an error when *winSize*, *nFilter*, *sampFreq*, *lowFreq* is less than or equal to 0, or when *nLinFilter* is less than or equal to 1, or *nLinFilter* is greater than *nFilter*.

ippStsFBankFreqErr	Indicates an error when *highLinFreq* is less than *lowFreq*, or *highFreq* is less than *highLinFreq*, or *highFreq* is greater than *sampFreq/2*, or *(highLinFreq > lowFreq)&&(nLinFilter==0)*, or *(highLinFreq < highFreq)&&(nLinFilter==nFilter)*.
ippStsFBankFlagErr	Indicates an error when the *mode* value is incorrect.
ippStsMemAllocErr	Indicates an error when no memory was allocated.

EmptyFBankInitAlloc

Initializes an empty filter bank structure.

```
IppStatus ippsEmptyFBankInitAlloc_16s(IppsFBankState_16s** pFBank, int*
   pFFTLen, int winSize, int nFilter, IppMelMode mode);
IppStatus ippsEmptyFBankInitAlloc_32f(IppsFBankState_32f** pFBank, int*
   pFFTLen, int winSize, int nFilter, IppMelMode mode);
```

Arguments

pFBank	Pointer to the filter bank structure to be created.
pFFTLen	Pointer to the length of FFT (*N = pFFTLen*[0]) used for the filter bank evaluation.
winSize	Frame length (in samples).
nFilter	Number of filters banks *K*.
mode	Flag determining the function's execution mode; can have the following value: IPP_POWER_SPECTRUM – indicates that the FFT power spectrum is used during the filter bank analysis.

Discussion

The function ippsEmptyFBankInitAlloc initializes the filter bank structure for the *nFilter* filters. The filter bank center frequencies can be set by the function ippsFBankSetCenters and the filter weights can be set by the function ippsFBankSetCoeffs.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pFFT Len* pointer is null.
ippStsSizeErr	Indicates an error when *winSize* or *nFilter* or *pFBank* is less than or equal to 0.
ippStsMemAllocErr	Indicates an error when no memory was allocated.

FBankFree

Destroys the structure for the filter bank analysis.

```
IppStatus ippsFBankFree_16s(IppsFBankState_16s* pFBank);
IppStatus ippsFBankFree_32f(IppsFBankState_32f* pFBank);
```

Arguments

pFBank	Pointer to the filter bank structure.

Discussion

The function ippsFBankFree destroys the filter bank structure and frees all memory associated with it.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* pointer is null.

FBankGetCenters

Retrieves the center frequencies of the triangular filter banks.

```
IppStatus ippsFBankGetCenters_16s(const IppsFBankState_16s* pFBank, int*
   pCenters);
IppStatus ippsFBankGetCenters_32f(const IppsFBankState_32f* pFBank, int*
   pCenters);
```

Arguments

pFBank	Pointer to the filter bank structure.
pCenters	Pointer to the output vector that contains the center frequencies.

Discussion

The function `ippsGetCenters` retrieves the indexes y_k (in FFT domain points) of the filter bank centers. The resulting array *pCenters* is of length *nFilter*+1. The filter bank structure *pFBank* must be initialized by either `ippsMelFBankInitAlloc` or `ippsMelLinFBankInitAlloc` function.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pFBank* or *pCenters* pointer is null.
`ippStsFBankErr`	Indicates an error when filter centers are not valid after filter bank initialization by `ippsEmptyFBankInitAlloc` function.

FBankSetCenters

Sets the center frequencies of the triangular filter banks.

```
IppStatus ippsFBankSetCenters_16s(IppsFBankState_16s* pFBank, const int*
   pCenters);
IppStatus ippsFBankSetCenters_32s(IppsFBankState_32s* pFBank, const int*
   pCenters);
```

Arguments

pFBank	Pointer to the filter bank structure.
pCenters	Pointer to the vector that contains center frequencies.

Discussion

The function `ippsSetCenters` sets the filter center indexes y_k in *pFBank*. The indexes must meet the following conditions:

$$0 \le ... \le y_k \le y_{k+1} \le ... \le N/2 , \quad i = 0,...K .$$

If the k-th center is modified, the filter weight coefficients in the (k-1)-th, k-th and (k+1)-th filter banks must be also modified using the `ippsFBankSetCoeffs` function.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pFBank* or *pCenters* pointer is null.

FBankGetCoeffs

Retrieves the filter bank weight coefficients.

```
IppStatus ippsFBankGetCoeffs_16s(const IppsFBankState_16s* pFBank, int fIdx,
   Ipp32f* pCoeffs);
IppStatus ippsFBankGetCoeffs_32f(const IppsFBankState_32f* pFBank, int fIdx,
   Ipp32f* pCoeffs);
```

Arguments

pFBank	Pointer to the filter bank structure
fIdx	Filter index.
pCoeffs	Pointer to the output vector that contains the filter coefficients.

Discussion

The function ippsFBankGetCoeffs copies the weight coefficients of the filter bank *fIdx* to the array *pCoeffs* which is of length $(y_{k+1} - y_{k-1} + 1)$.

Note that for ippsFBankGetCoeffs_16s function flavor, the filter output weights are represented in floating-point format.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pCoeffs* pointer is null.
ippStsSizeErr	Indicates an error when *fIdx* is less then 1 or greater then *nFilter*.
ippStsFBankErr	Indicates an error when *fIdx* filter coefficients are not available or valid.

FBankSetCoeffs

Sets the filter bank weight coefficients.

```
IppStatus ippsFBankSetCoeffs_16s(IppsFBankState_16s* pFBank, int fIdx, const
   Ipp32f* pCoeffs);
IppStatus ippsFBankSetCoeffs_32f(IppsFBankState_32f* pFBank, int fIdx, const
   Ipp32f* pCoeffs);
```

Arguments

pFBank	Pointer to the filter bank structure.
fIdx	Filter index.
pCoeffs	Pointer to the output coefficients vector.

Discussion

The function ippsFBankGetCoeffs sets the weight coefficients of the filter bank *fIdx*. The vector *pCoeffs* contains the weight coefficients from y_{k-1} to y_{k+1}.

Note that for ippsFBankSetCoeffs_16s function flavor, the weight coefficients are represented in floating-point format and are saturated to the [-1,1] interval.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pCoeffs* pointer is null.
ippStsSizeErr	Indicates an error when *fIdx* is less then 1 or greater then *nFilter*.
ippStsFBankErr	Indicates an error when the weight coefficients are not available or valid.

EvalFBank

Performs the filter bank analysis

```
IppStatus ippsEvalFBank_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
   IppsFBankState_16s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pDst, const
   IppsFBankState_16s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
   IppsFBankState_32f* pFBank);
```

Arguments

pSrc	Pointer to the source vector $[2^{pFFTOrder[0]}]$.
pDst	Pointer to the filter bank coefficients vector [*nFilter*].
pFBank	Pointer to the filter bank structure.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsEvalFBank performs the filter bank analysis for the input vector *pSrc*.

The execution mode set during the filter bank structure initialization determines the use of the input vector as follows:

If the IPP_POWER_SPECTRUM flag is set, the source vector is the wave signals. The magnitude of input signal spectrum is calculated for the filter bank analysis as follows:

$$x_k = \left| \left(\sum_{i=0}^{N-1} pSrc[i] \cdot \exp\left(-j \cdot 2\pi \frac{ik}{N}\right) \right) \right| , \quad 0 \le k \le N/2.$$

Otherwise, the input vector is used directly for filter bank analysis:

$x_k = pSrc[i] , \quad 0 \le k \le N/2.$

Depending on the mode, either formula (8.1) or (8.2) is used to obtain the filter bank coefficients. The input vector *pSrc* is destroyed after the analysis.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsFBankErr	Indicates an error when *pFBank* structure is not ready for calculation.

DCTLifterInitAlloc

Initializes the structure to perform DCT and lift the DCT coefficients.

```
IppStatus ippsDCTLifterInitAlloc_16s(IppsDCTLifterState_16s** pDCTLifter,
   int lenDCT, int lenCeps, int nLifter, Ipp32f val);
IppStatus ippsDCTLifterInitAlloc_C0_16s(IppsDCTLifterState_16s** pDCTLifter,
   int lenDCT, int lenCeps, int nLifter, Ipp32f val, Ipp32f val0);
IppStatus ippsDCTLifterInitAlloc_Mul_16s(IppsDCTLifterState_16s** pDCTLifter,
   int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_MulC0_16s(IppsDCTLifterState_16s**
   pDCTLifter, int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_32f(IppsDCTLifterState_32f** pDCTLifter,
   int lenDCT, int lenCeps, int nLifter, Ipp32f val);
IppStatus ippsDCTLifterInitAlloc_C0_32f(IppsDCTLifterState_32f** pDCTLifter,
   int lenDCT, int lenCeps, int nLifter, Ipp32f val, Ipp32f val0);
IppStatus ippsDCTLifterInitAlloc_Mul_32f(IppsDCTLifterState_32f** pDCTLifter,
   int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_MulC0_32f(IppsDCTLifterState_32f**
   pDCTLifter, int lenDCT, const Ipp32f* pLifter, int lenCeps);
```

Arguments

pDCTLifter	Pointer to the structure to be created for the DCT calculation and liftering.
lenDCT	Length of the DCT.

lenCeps	Number of the output coefficients (not including C_0).
nLifter	Liftering factor.
pLifter	Pointer to the liftering coefficients vector.
val	The scale factor for the output coefficients (except C_0).
val0	The scale factor for C_0.

Discussion

The function ippsDCTLifterInitAlloc initializes the structure for the DCT calculation and liftering.

The first DCT coefficient C_0 is usually ignored. Therefore, the output of the ippsDCTLifter function contains only C_1 to $C_{lenCeps}$ coefficients. However, if the C0 suffix is specified, C_0 is stored as the last element of the output vector.

The liftering coefficients are calculated according to the below formulae:

For functions without both the Mul and C0 suffixes,

$$l_i = \left(1 + \frac{nLifter}{2} \cdot \sin\left(\frac{\pi \cdot i}{nLifter}\right)\right) \cdot val \ , \ i = 1,\ldots lenCeps$$

For functions with the C0 suffix,

$$l_0 = val0$$
$$l_i = \left(1 + \frac{nLifter}{2} \cdot \sin\left(\frac{\pi \cdot i}{nLifter}\right)\right) \cdot val \ , \ i = 1,\ldots lenCeps$$

For functions with the Mul suffix,

$$l_i = pLifter[i-1] \ , \ i = 1,\ldots lenCeps$$

For functions that have both the Mul and C0 suffixes:

$$l_0 = pLifter[lenCeps-1]$$
$$l_i = pLifter[i-1] \ , \ i = 1,\ldots lenCeps$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pLifter* pointer is null.

ippStsSizeErr	Indicates an error when *lenDCT*, *lenCeps*, or *nLifter* is less than or equal to 0, or when *lenDCT* is less than *lenCeps*.
ippStsMemAllocErr	Indicates an error when no memory was allocated.

DCTLifterFree

Destroys the structure used for the DCT and liftering.

```
IppStatus ippsDCTLifterFree_16s(IppsDCTLifterState_16s* pDCTLifter);
IppStatus ippsDCTLifterFree_32f(IppsDCTLifterState_32f* pDCTLifter);
```

Arguments

pDCTLifter	Pointer to the DCT and liftering structure.

Discussion

The function ippsDCTLifterFree closes the DCT and liftering structure by freeing all memory associated with it.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDCTLifter* pointer is null.

DCTLifter

Performs the DCT and lifts the DCT coefficients.

```
IppStatus ippsDCTLifter_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
   IppsDCTLifterState_16s* pDCTLifter, int scaleFactor);
IppStatus ippsDCTLifter_32s16s_Sfs (const Ipp32s* pSrc, Ipp16s* pDst, const
   IppsDCTLifterState_16s* pDCTLifter, int scaleFactor);
```

```
IppStatus ippsDCTLifter_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
   IppsDCTLifterState_32f* pDCTLifter);
```

Arguments

pSrc	Pointer to the source vector [*lenDCT*].
pDst	Pointer to the output vector [*lenCeps*] or [*lenCeps*+1].
pDCTLifter	Pointer to the DCT and liftering structure.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsDCTLifter first performs the DCT and then lifts the DCT coefficients.

The DCT coefficients are calculated according to the formula:

$$y_i = \sum_{j=1}^{lenDCT} pSrc[j-1] \cdot \cos\left(\frac{\pi \cdot i \cdot (j-0.5)}{lenDCT}\right), \quad i = 0,\ldots lenCeps$$

The output coefficients are weighted as follows:

$$pDst[i-1] = l_i \cdot y_i, \quad i = 1,\ldots lenCeps$$

If the C_0 coefficient is required (*pDCTLifter* is initialized by functions with the C0 suffix), it is stored as the last element of the output vector:

$$pDst[lenCeps] = l_0 \cdot y_0$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc, pDst,* or *pFBank* pointer is null.

Example 8-1 MFCC feature calculation

```
/* Input:       samples[]          Input samples
                sample_number      Number of samples
   Output:      mfccs[][12]        The resulting MFCC coefficients
```

```
*/
void Calc_MFCC (Ipps32f *samples, int sample_number, ipps32f **mfccs) {
   Ipp32f* frame_buffer,fbank_buffer;
   IppsFBankState_32f *fbank;
   IppsDCTLifterState_32f *dctl;
   int fft_len,fft_order;
   int i,j;

   /* Initialize the structures */
   ippsMelFBankInitAlloc_32f(&fbank,       /* return the structure pointer */
                             &fft_order,   /* return the FFT length */
                             400,          /* 25ms window/512 point FFT */
                             16000,        /* sample rate */
                             64,           /* lowest frequency of interest */
                             8000,         /* highest frequency of interest */
                             24,           /* number of filter banks */
                             1126.994,     /* mel-scale factor 1 */
                             700,          /* mel-scale factor 2 */
                             IPP_FBANK_MELWGT | IPP_POWER_SPECTRUM);
   IppsDCTLifterInitAlloc_32f(&dctl,       /* return the structure pointer */
                             24,           /* filter bank channels */
                             12,           /* number of MFCC coefficients */
                             22,           /* liftering */
                             1.0);         /* no scaling */
   fft_len=1<<fft_order;
   frame_buffer=ippsMalloc_32f(fft_len);
   fbank_buffer=ippsMalloc_32f(24);
   /* Calculate MFCC features */
   for (i=j=0; i+400<sample_num; i+=160,j++) {
         /* Organize the input wave data into a frame */
         ippsCopyWithPadding_32f(&samples[i],400,frame_buffer,fft_len);
         /* Pre-emphasize the input signal with factor 0.97 */
         ippsPreemphasize_32f(frame_buffer,400,0.97);
         fft_buffer[0]*=(1.0-0.97);
         /* Add the hamming window to the input signal */
         ippsWinHamming_32fc_I(frame_buffer,400);
         /* Perform the filter bank analysis */
         ippsEvalFBank_32f(frame_buffer,fbank_buffer,fbank);
         /* Perform the DCT analysis and liftering */
         ippsDCTLifter_32f(fbank_buffer,mfccs[j],dctl);
```

```
    }
    /* Destroy the structures after calculation */
    ippsFree(fbank_buffer);
    ippsFree(frame_buffer);
    ippsFBankFree(fbank);
    ippsDCTLifterFree(dctl);
}
```

NormEnergy

Normalizes a vector of energy values.

```
IppStatus ippsNormEnergy_32f(Ipp32f* pSrcDst, int step, int height,
   Ipp32f silFloor, Ipp32f enScale);
IppStatus ippsNormEnergy_16s(Ipp16s* pSrcDst, int step, int height,
   Ipp16s silFloor, Ipp16s val, Ipp32f enScale);
IppStatus ippsNormEnergy_RT_32f(Ipp32f* pSrcDst, int step, int height,
   Ipp32f silFloor, Ipp32f maxE, Ipp32f enScale);
IppStatus ippsNormEnergy_RT_16s(Ipp16s* pSrcDst, int step, int height,
   Ipp16s silFloor, Ipp16s maxE, Ipp16s val, Ipp32f enScale);
```

Arguments

pSrcDst	Pointer to the input and output vector [*height***step*].
step	Sample step in the vector *pSrcDst*.
height	Number of samples for the normalization.
silFloor	Silence floor value.
val	Coefficient value.
maxE	Maximum energy value.
enScale	Energy scale.

Discussion

The function ippsNormEnergy normalizes the input vector that contains energy values. The normalization is performed as follows:
For functions without the RT suffix, the maximum energy value is calculated as

$$maxE = \max_{0 < i < height - 1} pSrcDst[i \cdot step]$$

For all functions, assuming $val = 1$ if not specified,

$minE = maxE - (silFloor \cdot \ln 10)/10$

$pSrcDst[i \cdot step] = val - (maxE - \max(pSrcDst[i \cdot step], minE)) \cdot enScale$,

for $0 \le i < height$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcDst* pointer is null.
ippStsSizeErr	Indicates an error when *step* or *height* is less than or equal to 0.

SumMeanVar

Calculates both the sum of a the vector and its square sum.

```
IppStatus ippsSumMeanVar_32f(const Ipp32f* pSrc, int srcStep, int height,
   Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsSumMeanVar_32f_I(const Ipp32f* pSrc, int srcStep, int height,
   Ipp32f* pSrcDstMean, Ipp32f* pSrcDstVar, int width);
IppStatus ippsSumMeanVar_16s32f(const Ipp16s* pSrc, int srcStep, int height,
   Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsSumMeanVar_16s32f_I(const Ipp16s* pSrc, int srcStep, int height,
   Ipp32f* pSrcDstMean, Ipp32f* pSrcDstVar, int width);
```

```
IppStatus ippsSumMeanVar_16s32s_Sfs(const Ipp16s* pSrc, int srcStep, int
   height, Ipp32s* pDstMean, Ipp32s* pDstVar, int width, int scaleFactor);
IppStatus ippsSumMeanVar_16s32s_ISfs(const Ipp16s* pSrc, int srcStep, int
   height, Ipp32s* pSrcDstMean, Ipp32s* pSrcDstVar, int width, int
   scaleFactor);
```

Arguments

pSrc	Pointer to the source vector [*height*srcStep*].
srcStep	Row step in the vector *pSrc* .
height	Number of rows in *pSrc*
pDstMean	Pointer to the destination vector that contains the sums [*width*].
pDstVar	Pointer to the destination vector that contains the square sums [*width*].
pSrcDstMean	Pointer to the source and destination vector that contains the sums [*width*].
pSrcDstVar	Pointer to the source and destination vector that contains the square sums [*width*].
width	Number of columns in the source vector *pSrc*.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsSumMeanVar calculates both the sums and the square sums of the source vectors as follows:

$$pDstVar[j] = \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] \cdot pSrc[i \cdot srcStep + j]$$

$$pDstMean[j] = \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j],$$

for $j = 0,\ldots width - 1$.

The function ippsSumMeanVar_I performs the in-place calculation as given by:

$$pSrcDstVar[j] += \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] \cdot pSrc[i \cdot srcStep + j]$$

$$pSrcDstMean[j] += \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j],$$

for $j = 0,\ldots width - 1$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDstMean*, *pDstVar*, *pSrcDstMean*, or *pSrcDstVar* pointer is null.
ippStsSizeErr	Indicates an error when *srcStep*, *width*, or *height* is less than or equal to 0, or when *width* is greater than *srcStep*.

NewVar

Calculates the variances given the sum and square sum accumulators.

```
IppStatus ippsNewVar_32f(const Ipp32f* pSrcMean, const Ipp32f* pSrcVar,
   Ipp32f* pDstVar, int width, Ipp32f val1, Ipp32f val2);
IppStatus ippsNewVar_32f_I(const Ipp32f* pSrcMean, Ipp32f* pSrcDstVar,
   int width, Ipp32f val1, Ipp32f val2);
IppStatus ippsNewVar_32s_Sfs(const Ipp32s* pSrcMean, const Ipp32s* pSrcVar,
   Ipp32s* pDstVar, int width, Ipp32f val1, Ipp32f val2, int scaleFactor);
IppStatus ippsNewVar_32s_ISfs(const Ipp32s* pSrcMean, Ipp32s* pSrcDstVar,
   int width, Ipp32f val1, Ipp32f val2, int scaleFactor);
```

Arguments

pSrcMean	Pointer to the vector that accumulates sums [*width*].
pSrcVar	Pointer to the vector that accumulates square sums[*width*].
pSrcDstVar	Pointer to the square sum accumulators and the resulting variance vector [*width*].
pDstVar	Pointer to the resulting variance vector [*width*].
width	Length of the variance vector *pDstVar*.
val1, val2	Constant coefficients.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsNewVar calculates the variance values given the sum and square sum accumulators as follows:

$pDstVar[i] = (pSrcVar[i] - pSrcMean[i] \cdot pSrcMean[i] \cdot val1) \cdot val2$, i=0...*width*-1

whereas the in-place function ippsNewVar_I uses the following formula:

$pSrcDstVar[i] = (pSrcDstVar[i] - pSrcMean[i] \cdot pSrcMean[i] \cdot val1) \cdot val2$,
for i = 0,...*width* - 1.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcMean*, *pSrcVar*, *pDstVar*, or *pSrcDstVar* pointer is null.
ippStsSizeErr	Indicates an error when *width* is less than or equal to 0.

RecSqrt

Calculates square roots of a vector and their reciprocals.

```
IppStatus ippsRecSqrt_32s_Sfs(Ipp32s* pSrcDst, int len, Ipp32s val, int
   scaleFactor);
IppStatus ippsRecSqrt_32f(Ipp32f* pSrcDst, int len, Ipp32f val);
```

Arguments

pSrcDst	Pointer to the source and destination vector [*len*].
len	Length of the vector *pSrcDst*.
val	Threshold for processed source values.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsRecSqrt calculates the square root of a vector and then takes the reciprocals. The operations are as follows:

$$pSrcDst[j] = \begin{cases} val, & if\ pSrcDst[j] < val \\ \dfrac{1}{\sqrt{pSrcDst[j]}}, & otherwise \end{cases}$$

for $j = 0,\ldots,len - 1$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0 or *val* is less than or equal to 0.
ippStsInvZero	Indicates a warning when all *pSrcDst*[i] are less than *val*.

Derivative Functions

Functions described in this section are used to calculate first and second derivatives of feature vectors. These functions process the sequence of input feature vectors and generate the corresponding output feature vectors that contain derivatives.

The input sequence $a_0, ..., a_{N-1}$ is stored as a one-dimensional array of length $N \cdot M$, where N is the number of feature vectors and M is the dimension of each feature a_i. Similarly, the output sequence $b_0, ..., b_{N-1}$ is also stored as a one-dimensional array of length $N \cdot K$, where K is the dimension of each feature b_j.
Certain constraints on values of M and K must be observed, specifically,

$K \geq M$;

$K \geq 2M$, for generating first derivatives;

$K \geq 3M$, for generating both first and second derivatives.

The `ippsCopyColumn` function copies the input sequence to the output sequence, that is, places the base features into the output sequence. The base features are placed in the first M elements of each output vector. The function `ippsEvalDelta` is then used to calculate the derivatives. While these two functions are used for general derivative calculation, the functions `ippsDelta` and `ippsDeltaDelta` provide combined but specific functionalities.

As the derivative operation accesses feature vectors both in the history and in the future, special treatment is required for the first `winSize` (delta window size) features, as well as for the last `winSize` features. For the first `winSize` features, the first feature vector is usually repeated to provide the history. For the last `winSize` features, the last feature vector is repeated to provide the future information.

CopyColumn

Copies the input sequence into the output sequence.

```
IppStatus ippsCopyColumn_16s_D2(const Ipp16s* pSrc, int srcWidth,
   Ipp16s* pDst, int dstWidth, int height);
IppStatus ippsCopyColumn_32f_D2(const Ipp32f* pSrc, int srcWidth,
   Ipp32f* pDst, int dstWidth, int height);
```

Arguments

pSrc	Pointer to the input feature sequence [*height*srcWidth*].
srcWidth	Length of each input feature vector.

pDst	Pointer to the output feature sequence [*height*dstWidth*].
dstWidth	Length of each output feature vector.
height	Number of features in the sequence.

Discussion

The function ippsCopyColumn copies the input feature sequence into the output sequence as follows:

$pDst[j \cdot dstWidth + i] = pSrc[j \cdot srcWidth + i]$, $0 \le i < srcWidth$, $0 \le j < height$

The unspecified elements in the output sequence remain unchanged.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *height* or *srcWidth* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *dstWidth* is less than *srcWidth*.

EvalDelta

Calculates the derivatives of feature vectors.

```
IppStatus ippsEvalDelta_16s_D2Sfs(Ipp16s* pSrcDst, int height, int step, int
   width, int offset, int winSize, Ipp16s val, int scaleFactor);
IppStatus ippsEvalDelta_32f_D2(Ipp32f* pSrcDst, int height, int step, int
   width, int offset, int winSize, Ipp32f val);
IppStatus ippsEvalDeltaMul_16s_D2Sfs(Ipp16s* pSrcDst, int height, int step,
   const Ipp16s* pVal, int width, int offset, int winSize, int scaleFactor);
IppStatus ippsEvalDeltaMul_32f_D2(Ipp32f* pSrcDst, int height, int step, const
   Ipp32f* pVal, int width, int offset, int winSize);
```

Arguments

pSrcDst	Pointer to the input and output sequence [*height*step*].
height	Number of features in *pSrcDst*.
step	Length of each feature in *pSrcDst*.
width	The number of derivatives to be calculated for each feature.
offset	Offset to place the derivative values.
winSize	The delta window size
val	The delta coefficient.
pVal	Pointer to the delta coefficients vector [*width*].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsEvalDelta calculates the derivatives for the input feature sequence. The base feature ranges from *offset* to (*offset* + *width* - 1) in each feature vector. The output derivatives are stored in each feature vector next to the base features ranging from (*offset* + *width*) to (*offset* + 2**width* - 1). The operations are detailed as follows:

For ippsEvalDelta functions,

$$pSrcDst[i \cdot step + width + j] = val \cdot \sum_{k=1}^{winSize} k \cdot \{pSrcDst[\min(i+k, height-1) \cdot step + j] - pSrcDst[\max(i-k, 0) \cdot step + j]\},$$

$$0 \le i < height, \quad offset \le j < offset + width .$$

For ippsEvalDeltaMul functions,

$$pSrcDst[i \cdot step + width + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pSrcDst[\min(i+k, height-1) \cdot step + j] - pSrcDst[\max(i-k, 0) \cdot step + j]\},$$

$0 \le i < height, \quad offset \le j < offset + width$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcDst* pointer is null.
ippStsSizeErr	Indicates an error when *height*, *width*, or *winSize* is less than or equal to 0; or *offset* is less than 0; or *height* is less than 2**winSize*.
ippStsStrideErr	Indicates an error when *step* is less than *offset+2*width*.

Delta

Copies the base features and calculates the derivatives of feature vectors.

```
IppStatus ippsDelta_Win1_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth, Ipp16s*
   pDst, int dstStep, int height, Ipp16s val, int deltaMode, int scaleFactor);
IppStatus ippsDelta_Win2_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth, Ipp16s*
   pDst, int dstStep, int height, Ipp16s val, int deltaMode, int scaleFactor);
IppStatus ippsDelta_Win1_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
   pDst, int dstStep, int height, Ipp32f val, int deltaMode);
IppStatus ippsDelta_Win2_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
   pDst, int dstStep, int height, Ipp32f val, int deltaMode);
IppStatus ippsDeltaMul_Win1_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pVal,
   int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode, int
   scaleFactor);
IppStatus ippsDeltaMul_Win2_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pVal,
   int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode, int
   scaleFactor);
IppStatus ippsDeltaMul_Win1_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal, int
   srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

```
IppStatus ippsDeltaMul_Win2_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal, int
   srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

Arguments

pSrc	Pointer to the input feature sequence [*height*srcWidth*].
srcWidth	Length of the feature vector in the input sequence *pSrc*.
pDst	Pointer to the output feature sequence
dstStep	Length of the feature vector in the output sequence *pDst*.
height	Number of feature vectors.
val	Delta coefficient.
deltaMode	Execution mode.
pVal	Pointer to the delta coefficients vector [*width*].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsDelta or ippsDeltaMul provides a combined functionality of ippsCopyColumn and ippsEvalDelta. First, the input feature vectors are copied to the output sequence. Then the derivatives are calculated.

In the subsequent discussion, the following conditions hold:

The function suffix Win1 or Win2 specifies the delta window size, *winSize* = 1 or 2, respectively;

The function ippsDelta implies the following constraints on the delta coefficients:

$pVal[j] \equiv val, \ \forall j.$

The execution mode *deltaMode* provides additional controls along the base feature copy and derivative calculation process. The admissible values of *deltaMode* and the corresponding function execution logic are the following:

1. *deltaMode* is equal to IPP_DELTA_BEGIN|IPP_DELTA_END

Perform the offline delta feature calculation. All base features are assumed available at the time of the calculation. The base features are copied from the input stream *pSrc* to the output stream *pDst* as follows:

$$pDst[i \cdot dstStep + j] = pSrc[i \cdot srcWidth + j], \quad 0 \le i < height, \quad 0 \le j < srcWidth. \tag{8.4}$$

Then the derivatives are calculated as given by:

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pDst[\min(i+k, height-1) \cdot dstStep + j] - pDst[\max(i-k, 0) \cdot dstStep + j]\}, \tag{8.5}$$

for $0 \le i < height, \quad 0 \le j < srcWidth$.

2. *deltaMode* is equal to 0

Perform the online delta feature calculation. The input features are the current segment of a continuous stream. The function ippsDelta calculates the partial delta features accordingly. First, the base features are copied as follows:

$$pDst[(i + 2 \cdot winSize) \cdot dstStep + j] = pSrc[i \cdot srcWidth + j], \tag{8.6}$$

for $0 \le i < height, \quad 0 \le j < srcWidth$.

Then the derivatives are calculated as given by:

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pDst[(i + k) \cdot dstStep + j] - pDst[(i - k) \cdot dstStep + j]\}, \tag{8.7}$$

for $0 \le i - winSize < height, \ 0 \le j < srcWidth$.

In this execution mode, the base features in $pDst[i \cdot dstStep + j]$ and the derivatives in $pDst[k \cdot dstStep + srcWidth + j]$ are assumed available through a previous derivative calculation for $0 \le j < srcWidth$, $0 \le i < 2 \cdot winSize$, and $0 \le k < winSize$.

3. *deltaMode* is equal to IPP_DELTA_BEGIN

Perform the partial online delta feature calculation, where the beginning of the input stream is known. First, the base features are copied as in equation (8.4). Then, the derivatives are calculated as follows:

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pDst[(i+k) \cdot dstStep + j] - pDst[\max(i-k, 0) \cdot dstStep + j]\}, \quad (8.8)$$

for $0 \le i < height - winSize$, $0 \le j < srcWidth$.

4. *deltaMode* is equal to IPP_DELTA_END

Perform the partial online delta feature calculation, where the ending of the input stream is known. First, the base features are copied as in equation (8.6). Then, the derivatives are calculated as follows:

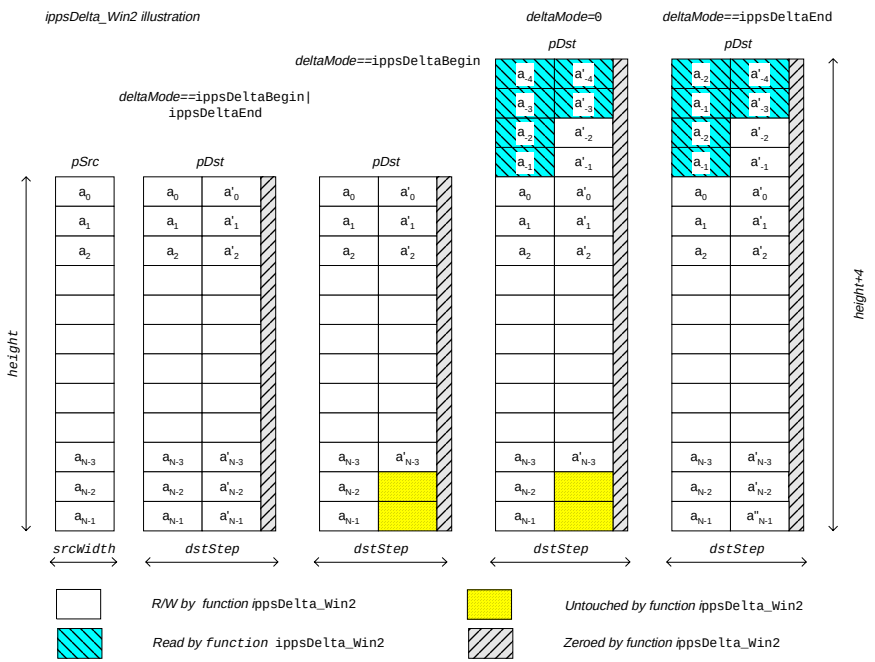
$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pDst[\min(i+k, height + 2 \cdot winSize - 1) \cdot dstStep + j] - pDst[(i-k) \cdot dstStep + j]\}$$

for $0 \le i - winSize < height + winSize$, $0 \le j < srcWidth$.

In this execution mode, the base features in $pDst[i \cdot dstStep + j]$ and the derivatives in $pDst[k \cdot dstStep + srcWidth + j]$ are assumed available through a previous derivative calculation for $0 \le j < srcWidth$, $0 \le i < 2 \cdot winSize$, and $0 \le k < winSize$.

The following figure illustrates the above four delta calculation modes:

Figure 8-1 Execution Modes of `ippsDelta_Win2` function



Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`*, *`pDst`*, or *`pVal`* pointer is null.
`ippStsSizeErr`	Indicates an error when *`srcWidth`* is less than or equal to 0; or *`height`* is less than or equal to 0; or *`height`* is less than or equal to *`winSize`*+1 when `IPP_DELTA_BEGIN` is set; or *`height`* is less than 0 when `IPP_DELTA_BEGIN` is not set.
`ippStsStrideErr`	Indicates an error when *`dstStep`* is less than 2**`srcWidth`*.

DeltaDelta

Copies the base features and calculates their first and second derivatives.

```
IppStatus ippsDeltaDelta_Win1_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth,
   Ipp16s* pDst, int dstStep, int height, Ipp16s val1, Ipp16s val2, int
   deltaMode, int scaleFactor);
IppStatus ippsDeltaDelta_Win2_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth,
   Ipp16s* pDst, int dstStep, int height, Ipp16s val1, Ipp16s val2, int
   deltaMode, int scaleFactor);
IppStatus ippsDeltaDelta_Win1_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
   pDst, int dstStep, int height, Ipp32f val1, Ipp32f val2, int deltaMode);
IppStatus ippsDeltaDelta_Win2_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
   pDst, int dstStep, int height, Ipp32f val1, Ipp32f val2, int deltaMode);
IppStatus ippsDeltaDeltaMul_Win1_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
   pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode,
   int scaleFactor);
IppStatus ippsDeltaDeltaMul_Win2_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
   pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode,
   int scaleFactor);
IppStatus ippsDeltaDeltaMul_Win1_32f_D2(const Ipp32f* pSrc, const Ipp32f*
   pVal, int srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
IppStatus ippsDeltaDeltaMul_Win2_32f_D2(const Ipp32f* pSrc, const Ipp32f*
   pVal, int srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

Arguments

pSrc	Pointer to the input feature sequence [*height*srcWidth*].
srcWidth	Length of the input feature in the input sequence *pSrc*.
pDst	Pointer to the output feature sequence.
dstStep	Length of the output feature in the output sequence *pDst*.
height	Number of feature vectors.
val1, val2	The first and second delta coefficients.

deltaMode	Execution mode.
pVal	Pointer to the delta coefficients vector [*width*].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsDeltaDelta or ippsDeltaDeltaMul provides a combined functionality of ippsCopyColumn and double operations of ippsEvalDelta. First, the input feature vectors are copied to the output sequence. Then the first and second derivatives are calculated.

In the subsequent discussion, the following conditions hold:

The function suffix Win1 or Win2 specifies the delta window size, *winSize* = 1 or 2, respectively;

The function ippsDeltaDelta implies the following constraints on the delta coefficients:

$pVal[j] \equiv val1, \quad \forall j$, for the first derivative calculation and

$pVal[j] \equiv val2, \quad \forall j$, for the second derivative calculation.

The execution mode *deltaMode* provides additional controls along the base feature copy and derivative calculation process. The admissible values of *deltaMode* and the corresponding function execution logic are the following:

1. *deltaMode* is equal to IPP_DELTA_BEGIN|IPP_DELTA_END

   Perform the offline delta-delta feature calculation. All base features are assumed available at the time of the calculation. First, the base features are copied from the input stream *pSrc* to the output stream *pDst* according to (8.4). Then the first derivatives are calculated as given by (8.5) for $0 \le i < height$, $0 \le j < srcWidth$. Finally, the second derivatives are calculated also by formula (8.5) for $0 \le i < height$, $0 \le j\text{-}srcWidth < srcWidth$.

2. *deltaMode* is equal to 0

   Perform the online delta-delta feature calculation. The input features are the current segment of a continuous stream. The function ippsDeltaDelta calculates the partial delta-delta features accordingly.

First, the base features are copied as follows:

$$pDst[(i+3\cdot winSize)\cdot dstStep+j] = pSrc[i\cdot srcWidth+j], \quad (8.9)$$

for $0 \le i < height$, $0 \le j < srcWidth$.

Then the first derivatives are calculated as given by (8.7), for $0 \le i - 2\cdot winSize < height$ and $0 \le j < srcWidth$.

Finally, the second derivatives are calculated also by formula (8.7) for $0 \le i - winSize < height$, $0 \le j - srcWidth < srcWidth$.

In this execution mode, the base features in $pDst[i\cdot dstStep+j]$, the first derivatives in $pDst[k\cdot dstStep+srcWidth+j]$, and the second derivatives in $pDst[l\cdot dstStep+2\cdot srcWidth+j]$ are assumed available through a previous delta-delta calculation for $0 \le j < srcWidth$, $0 \le i < 3\cdot winSize$, $0 \le k < 2\cdot winSize$, and $0 \le l < winSize$.

3. *deltaMode* is equal to IPP_DELTA_BEGIN

Perform the partial online delta-delta feature calculation, where the beginning of the input stream is known. Initially, the base features are copied as in equation (8.4). Then, the first derivatives are calculated as given by (8.8), for $0 \le i < height - winSize$ and $0 \le j < srcWidth$.

Finally, the second derivatives are calculated also by formula (8.8) for $0 \le i < height - 2\cdot winSize$ and $0 \le j - srcWidth < srcWidth$.

4. *deltaMode* is equal to IPP_DELTA_END

Perform the partial online delta-delta feature calculation, where the ending of the input stream is known. Initially, the base features are copied as in equation (8.9). Then, the first derivatives are calculated as follows:

$$pDst[i\cdot dstStep+srcWidth+j] = pVal[j]\cdot\sum_{k=1}^{winSize} k\cdot\{ \quad (8.10)$$

$$pDst[\min(i+k, height+3\cdot winSize-1)\cdot dstStep+j] - pDst[(i-k)\cdot dstStep+j]\}$$

for $0 \le i - 2\cdot winSize < height + winSize$, $0 \le j < srcWidth$.

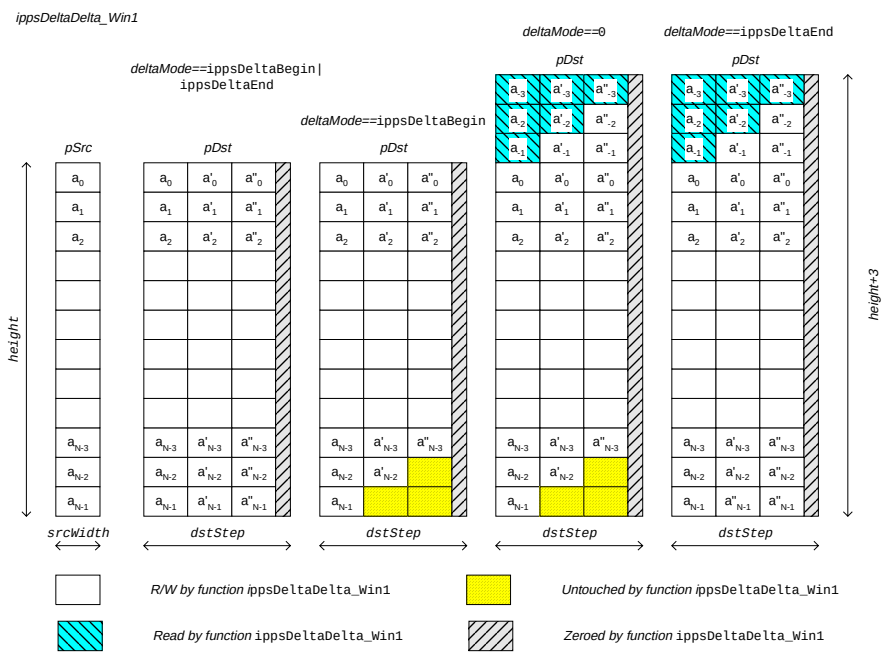
Finally, the second derivatives are calculated also according to (8.10) where the indexes *i* and *j* vary in the following ranges:

$0 \le i - winSize < height + 2 \cdot winSize$, $0 \le j - srcWidth < srcWidth$.

In this execution mode, the base features in $pDst[i \cdot dstStep + j]$, the first derivatives in $pDst[k \cdot dstStep + srcWidth + j]$, and the second derivatives in $pDst[l \cdot dstStep + 2 \cdot srcWidth + j]$ are assumed available through a previous delta-delta calculation for $0 \le j < srcWidth$, $0 \le i < 3 \cdot winSize$, $0 \le k < 2 \cdot winSize$, and $0 \le l < winSize$.

The following figure illustrates the above four delta-delta calculation modes:

Figure 8-2 Execution Modes of `ippsDeltaDelta_Win1` function



Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, or *pVal* pointer is null.
ippStsSizeErr	Indicates an error when *srcWidth* is less than or equal to 0; or *height* is less than or equal to 0; or *height* is less than or equal to 2*(*winSize*+1) when IPP_DELTA_BEGIN is set; or *height* is less than 0 when IPP_DELTA_BEGIN is not set.
ippStsStrideErr	Indicates an error when *dstStep* is less than 3**srcWidth*.

Model Evaluation

This section describes functions that evaluate the acoustic and language models.

AddNRows

Adds N vectors from a vector array.

```
IppStatus ippsAddNRows_32f_D2(Ipp32f* pSrc, int height, int offset,
   int step, Ipp32s* pInd, Ipp16u* pAddInd, int rows, Ipp32f* pDst,
   int width, Ipp32f weight);
```

Arguments

pSrc	Pointer to the vector array [*height***step*].
height	Number of rows in the vector array *pSrc*.
offset	Offset to the vector of interest in the vector array *pSrc*.
step	Row step in the vector array *pSrc*.
pInd	Pointer to the index vector [*rows*].
pAddInd	Pointer to the additional index vector [*rows*].
rows	Number of vectors to be added.

pDst	Pointer to the output vector [*width*].
width	Length of the output vector *pDst*.
weight	Weight value to be added to the output.

Discussion

The function ippsAddNRows calculates the sum of the *rows* number of vectors, according to the sum of the indexing vectors *pInd* and *pAddInd*, from the vector array *pSrc* as follows:

$$pDst[k] = weight + \sum_{i=0}^{rows} pSrc[k + offset + step \cdot (pInd[i] + pAddInd[i])],$$

$0 \le k < width$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pInd*, *pAddInd*, or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *height*, *width*, or *rows* is less than or equal to 0; or (*pInd*[*i*]+*pAddInd*[*i*])or *offset* is less than 0; or (*pInd*[*i*]+*pAddInd*[*i*]) is greater than or equal to *height*.
ippStsStrideErr	Indicates an error when *step* is less than *width* + *offset*.

ScaleLM

Scales vector elements with thresholding.

```
IppStatus ippsScaleLM_16s32s(const Ipp16s* pSrc, Ipp32s* pDst, int len, Ipp16s
   floor, Ipp16s scale, Ipp32s base);
IppStatus ippsScaleLM_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
   floor, Ipp32f scale, Ipp32f base);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pDst	Pointer to the output vector [*len*].
len	Length of the vector *pSrc* or *pDst*.
floor	Threshold value.
scale	Scaling factor.
base	Additive factor.

Discussion

The function ippsScaleLM sets threshold on the input vector *pSrc* and scales the vector elements as follows:

$$pDst[n] = scale \cdot \max(pSrc[n], floor) + base, \quad 0 \le n < len$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

LogAdd

Adds two vectors in the logarithmic representation.

```
IppStatus ippsLogAdd_32f(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len,
   IppHintAlgorithm hint);
IppStatus ippsLogAdd_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len,
   IppHintAlgorithm hint);
```

Arguments

pSrc	Pointer to the first input vector [*len*].
pSrcDst	Pointer to the second input vector and also the output vector [*len*].
len	Number of elements in the input and output vectors.
hint	Suggestion for using specific code for logarithmic addition.

Discussion

The function ippsLogAdd adds the *pSrc* and *pSrcDst* vectors, whose elements are in the logarithmic representation. The output vector *pSrcDst*, also in the logarithmic representation, is calculated as follows:

$$pSrcDst[i] = \ln(e^{pSrc[i]} + e^{pSrcDst[i]}), \quad 0 \le i < len$$

The *hint* argument suggests using special code which provides for faster but less accurate calculation, or more accurate but slower calculation. The values you can enter for the *hint* argument are listed in Table 7-2, "Flag and Hint Arguments".

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pSrcDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

LogSub

Subtracts a vector from another vector, in the logarithmic representation.

```
IppStatus ippsLogSub_32f(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsLogSub_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the first input vector [*len*].
pSrcDst	Pointer to the second input and the output vector [*len*].
len	Number of elements in the input and output vectors.

Discussion

The function ippsLogSub subtracts the vector *pSrcDst* from the vector *pSrc*, both of them in the logarithmic representation. The output vector *pSrcDst* is calculated as follows:

$$pSrcDst[i] = \ln(e^{pSrc[i]} - e^{pSrcDst[i]}), \quad 0 \le i < len$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pSrcDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsNoOperation	Indicates an error when *pSrc*[*i*] is less than *pSrcDst*[*i*].

MahDistSingle

Calculates the Mahalanobis distance for a single observation vector.

```
IppStatus ippsMahDistSingle_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s*
   pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, int scaleFactor);
IppStatus ippsMahDistSingle_16s32f(const Ipp16s* pSrc, const Ipp16s* pMean,
   const Ipp16s* pVar, int len, Ipp32f* pResult);
IppStatus ippsMahDistSingle_32f(const Ipp32f* pSrc, const Ipp32f* pMean, const
   Ipp32f* pVar, int len, Ipp32f* pResult);
IppStatus ippsMahDistSingle_64f(const Ipp64f* pSrc, const Ipp64f* pMean, const
   Ipp64f* pVar, int len, Ipp64f* pResult);
```

```
IppStatus ippsMahDistSingle_32f64f(const Ipp32f* pSrc, const Ipp32f* pMean,
   const Ipp32f* pVar, int len, Ipp64f* pResult);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pMean	Pointer to the mean vector [*len*].
pVar	Pointer to the variance vector [*len*].
len	Number of elements in the input, mean, and variance vectors.
pResult	Pointer to the result.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function `ippsMahDistSingle` calculates the Mahalanobis distance for the input vector *pSrc*, given the mean vector *pMean* and the variance vector *pVar*. The calculation is as follows:

$$pResult[0] = \sum_{i=0}^{len-1} pVar[i] \cdot (pSrc[i] - pMean[i])^2$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *pMean*, *pVar*, or *pResult* pointer is null.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

MahDist

Calculates the Mahalanobis distances for multiple observation vectors.

```
IppStatus ippsMahDist_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
   pMean, const Ipp32f* pVar, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDist_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean, const
   Ipp32f* pVar, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDist_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
   pMean, const Ipp64f* pVar, int width, Ipp64f* pDst, int height);
IppStatus ippsMahDist_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean, const
   Ipp64f* pVar, int width, Ipp64f* pDst, int height);
```

Arguments

pSrc	Pointer to the input vector [*height***step*].
mSrc	Pointer to the input matrix [*height*][*width*].
step	Row step in the input vector *pSrc*.
pMean	Pointer to the mean vector [*width*].
pVar	Pointer to the variance vector [*width*].
width	Length of the mean and variance vectors.
pDst	Pointer to the result vector [*height*].
height	Number of rows in the input matrix *mSrc*.

Discussion

The function ippsMahDist calculates the Mahalanobis distances for multiple input vectors, given the mean vector *pMean* and the variance vector *pVar*. The calculation is as follows:
For functions with the D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2, \quad 0 \le i < height.$$

For functions with the D2L suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2 \ , 0 \le i < height.$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pMean*, *pVar*, or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

MahDistMultiMix

Calculates the Mahalanobis distances for multiple means and variances .

```
IppStatus ippsMahDistMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f* pVar,
   int step, const Ipp32f* pSrc, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDistMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
   mVar, const Ipp32f* pSrc, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDistMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f* pVar,
   int step, const Ipp64f* pSrc, int width, Ipp64f* pDst, int height);
IppStatus ippsMahDistMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
   mVar, const Ipp64f* pSrc, int width, Ipp64f* pDst, int height);
```

Arguments

pMean	Pointer to the mean vector [*height***step*].
pVar	Pointer to the variance vector [*height***step*].
mMean	Pointer to the mean matrix [*height*][*width*].

mVar	Pointer to the variance matrix [*height*][*width*].
step	Row step in mean and variance vectors.
pSrc	Pointer to the input vector [*width*].
width	Length of the input vector *pSrc*.
pDst	Pointer to the result vector [*height*].
height	Length of the result vector *pDst*.

Discussion

The function ippsMahDistMultiMix calculates the Mahalanobis distances for a single observation vector but multiple mean and variance pairs as follows:

For functions with the D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2, \quad 0 \le i < height.$$

For functions with the D2L suffix,

$$pDst[i] = \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2, \quad 0 \le i < height.$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mMean*, *mVar*, *pMean*, *pVar*, or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussSingle

Calculates the observation probability for a single Gaussian with an observation vector.

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussSingle_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s*
   pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, Ipp32s val,
   int scaleFactor);
IppStatus ippsLogGaussSingle_16s32f(const Ipp16s* pSrc, const Ipp16s* pMean,
   const Ipp16s* pVar, int len, Ipp32f* pResult, Ipp32f val);
IppStatus ippsLogGaussSingle_32f(const Ipp32f* pSrc, const Ipp32f* pMean,
   const Ipp32f* pVar, int len, Ipp32f* pResult, Ipp32f val);
IppStatus ippsLogGaussSingle_32f64f(const Ipp32f* pSrc, const Ipp32f* pMean,
   const Ipp32f* pVar, int len, Ipp64f* pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_64f(const Ipp64f* pSrc, const Ipp64f* pMean,
   const Ipp64f* pVar, int len, Ipp64f* pResult, Ipp64f val);
```

Case 2: Operation for the diagonal covariance matrix

```
IppStatus ippsLogGaussSingle_DirectVar_16s32s_Sfs(const Ipp16s* pSrc, const
   Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, Ipp32s val,
   int scaleFactor);
IppStatus ippsLogGaussSingle_DirectVar_16s32f(const Ipp16s* pSrc, const
   Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32f* pResult, Ipp32f val);
IppStatus ippsLogGaussSingle_DirectVar_32f(const Ipp32f* pSrc, const Ipp32f*
   pMean, const Ipp32f* pVar, int len, Ipp32f* pResult, Ipp32f val);
IppStatus ippsLogGaussSingle_DirectVar_32f64f(const Ipp32f* pSrc, const
   Ipp32f* pMean, const Ipp32f* pVar, int len, Ipp64f* pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_DirectVar_64f(const Ipp64f* pSrc, const Ipp64f*
   pMean, const Ipp64f* pVar, int len, Ipp64f* pResult, Ipp64f val);
```

Case 3: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussSingle_IdVar_16s32s_Sfs(const Ipp16s* pSrc, const
   Ipp16s* pMean, int len, Ipp32s* pResult, Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussSingle_IdVar_16s32f(const Ipp16s* pSrc, const Ipp16s*
   pMean, int len, Ipp32f* pResult, Ipp32f val);
```

```
IppStatus ippsLogGaussSingle_IdVar_32f(const Ipp32f* pSrc, const Ipp32f*
   pMean, int len, Ipp32f* pResult, Ipp32f val);
IppStatus ippsLogGaussSingle_IdVar_32f64f(const Ipp32f* pSrc, const Ipp32f*
   pMean, int len, Ipp64f* pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_IdVar_64f(const Ipp64f* pSrc, const Ipp64f*
   pMean, int len, Ipp64f* pResult, Ipp64f val);
```

Case 4: Operation for the block diagonal covariance matrix

```
IppStatus ippsLogGaussSingle_BlockDVar_16s32s_Sfs(const Ipp16s* pSrc, const
   Ipp16s* pMean, const IppsBlockDMatrix_16s * pBlockVar, int len, Ipp32s*
   pResult, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussSingle_BlockDVar_16s32f(const Ipp16s* pSrc, const
   Ipp16s* pMean, const IppsBlockDMatrix_16s* pBlockVar, int len, Ipp32f*
   pResult, Ipp32f val);
IppStatus ippsLogGaussSingle_BlockDVar_32f(const Ipp32f* pSrc, const Ipp32f*
   pMean, const IppsBlockDMatrix_32f* pBlockVar, int len, Ipp32f* pResult,
   Ipp32f val);
IppStatus ippsLogGaussSingle_BlockDVar_32f64f(const Ipp32f* pSrc, const
   Ipp32f* pMean, const IppsBlockDMatrix_32f * pBlockVar, int len, Ipp64f*
   pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_BlockDVar_64f(const Ipp64f* pSrc, const Ipp64f*
   pMean, const IppsBlockDMatrix_64f * pBlockVar, int len, Ipp64f* pResult,
   Ipp64f val);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pMean	Pointer to the mean vector [*len*].
pVar	Pointer to the variance vector [*len*].
pBlockVar	Pointer to the block diagonal variance matrix.
len	Number of elements in the input, mean, and variance vectors.
pResult	Pointer to the result.
val	Gaussian constant.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLogGaussSingle calculates the observation probability for a single observation vector and a Gaussian mixture component. The result *pResult* is in the logarithmic representation.
For functions without the DirectVar, IdVar, or BlockDVar suffix, the covariance matrix is assumed to be inverse diagonal:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} pVar[i] \cdot (pSrc[i] - pMean[i])^2$$

For functions with the DirectVar suffix, the covariance matrix is assumed to be diagonal:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} (pSrc[i] - pMean[i])^2 / pVar[i]$$

For functions with the IdVar suffix, the covariance matrix is assumed to be identity:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} (pSrc[i] - pMean[i])^2$$

For functions with the BlockDVar suffix, the covariance matrix is assumed to be block diagonal:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} \sum_{j=0}^{len-1} (pSrc[i] - pMean[i]) \cdot V[i][j] \cdot (pSrc[j] - pMean[j]),$$

where $V[i][j]$ is the element of the block diagonal matrix *pVar*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pMean*, *pVar*, *pBlockVar*, or *pResult* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

LogGauss

Calculates the observation probability for a single Gaussian with multiple observation vectors.

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGauss_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int height,
   Ipp32s val, int scaleFactor);

IppStatus ippsLogGauss_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s* pMean,
   const Ipp16s* pVar, int width, Ipp32s* pDst, int height, Ipp32s val, int
   scaleFactor);

IppStatus ippsLogGauss_16s32f_D2(const Ipp16s* pSrc, int step, const Ipp16s*
   pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height, Ipp32f val);

IppStatus ippsLogGauss_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s* pMean,
   const Ipp16s* pVar, int width, Ipp32f* pDst, int height, Ipp32f val);

IppStatus ippsLogGauss_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
   pMean, const Ipp32f* pVar, int width, Ipp32f* pDst, int height, Ipp32f val);

IppStatus ippsLogGauss_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean, const
   Ipp32f* pVar, int width, Ipp32f* pDst, int height, Ipp32f val);

IppStatus ippsLogGauss_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
   pMean, const Ipp64f* pVar, int width, Ipp64f* pDst, int height, Ipp64f val);

IppStatus ippsLogGauss_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean, const
   Ipp64f* pVar, int width, Ipp64f* pDst, int height, Ipp64f val);
```

Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGauss_IdVar_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
   Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int
   scaleFactor);

IppStatus ippsLogGauss_IdVar_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
   pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int scaleFactor);

IppStatus ippsLogGauss_IdVar_16s32f_D2(const Ipp16s* pSrc, int step, const
   Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val);

IppStatus ippsLogGauss_IdVar_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
   pMean, int width, Ipp32f* pDst, int height, Ipp32f val);
```

```
IppStatus ippsLogGauss_IdVar_32f_D2(const Ipp32f* pSrc, int step, const
   Ipp32f* pMean, int width, Ipp32f* pDst, int height, Ipp32f val);
IppStatus ippsLogGauss_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean,
   int width, Ipp32f* pDst, int height, Ipp32f val);
IppStatus ippsLogGauss_IdVar_64f_D2(const Ipp64f* pSrc, int step, const
   Ipp64f* pMean, int width, Ipp64f* pDst, int height, Ipp64f val);
IppStatus ippsLogGauss_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
   int width, Ipp64f* pDst, int height, Ipp64f val);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
step	Row step in the input vector *pSrc*.
pMean	Pointer to the mean vector [*width*].
pVar	Pointer to the variance vector [*width*].
width	Length of the mean and variance vectors.
pDst	Pointer to the result vector [*height*].
height	Number of rows in the input matrix, also the length of the result vector *pDst*.
val	Gaussian constant.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLogGauss calculates the observation probability for multiple observation vectors for a Gaussian mixture component. The result *pResult* is in the logarithmic representation.

For functions without the IdVar suffix, the covariance matrix is assumed to be inverse diagonal:

For functions with the additional D2 suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2, \; 0 \le i < height$$

For functions with the additional `D2L` suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2, \; 0 \le i < height.$$

For functions with the `IdVar` suffix, the covariance matrix is assumed to be identity:

For functions with the additional `D2` suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2, \; 0 \le i < height$$

For functions with the additional `D2L` suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2, \; 0 \le i < height.$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *mSrc*, *pMean*, *pVar*, or *pDst* pointer is null.
`ippStsSizeErr`	Indicates an error when *width* or *height* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *step* is less than *width*.

LogGaussMultiMix

Calculates the observation probability for multiple Gaussian mixture components.

```
IppStatus ippsLogGaussMultiMix_16s32s_D2Sfs(const Ipp16s* pMean, const Ipp16s*
   pVar, int step, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height,
   int scaleFactor);
IppStatus ippsLogGaussMultiMix_16s32s_D2LSfs(const Ipp16s** mMean, const
   Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height,
   int scaleFactor);
IppStatus ippsLogGaussMultiMix_16s32f_D2(const Ipp16s* pMean, const Ipp16s*
   pVar, int step, const Ipp16s* pSrc, int width, Ipp32f* pSrcDst, int height);
IppStatus ippsLogGaussMultiMix_16s32f_D2L(const Ipp16s** mMean, const Ipp16s**
   mVar, const Ipp16s* pSrc, int width, Ipp32f* pSrcDst, int height);
IppStatus ippsLogGaussMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f* pVar,
   int step, const Ipp32f* pSrc, int width, Ipp32f* pSrcDst, int height);
IppStatus ippsLogGaussMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
   mVar, const Ipp32f* pSrc, int width, Ipp32f* pSrcDst, int height);
IppStatus ippsLogGaussMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f* pVar,
   int step, const Ipp64f* pSrc, int width, Ipp64f* pSrcDst, int height);
IppStatus ippsLogGaussMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
   mVar, const Ipp64f* pSrc, int width, Ipp64f* pSrcDst, int height);
```

Arguments

pMean	Pointer to the mean vector [*height***step*].
pVar	Pointer to the variance vector [*height***step*].
mMean	Pointer to the mean matrix [*height*][*width*].
mVar	Pointer to the variance matrix [*height*][*width*].
step	Row step in the mean and variance vectors.
pSrc	Pointer to the input vector [*width*].
pSrcDst	Pointer to the result vector [*height*].
width	Number of columns in the mean matrix *mMean*.

height	Number of rows in the mean matrix *mMean*.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLogGaussMultiMix calculates the observation probability for multiple Gaussian mixture components. The results are in the logarithmic representation. The calculations are as follows:

For functions with the D2 suffix,

$$pSrcDst[i] = pSrcDst[i] - 0.5\sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2$$

$0 \le i < height$.

For functions with the D2L suffix,

$$pSrcDst[i] = pSrcDst[i] - 0.5\sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 \quad ,$$

$0 \le i < height$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *mMean*, *mVar*, *pMean*, *pVar*, or *pSrcDst* pointer is null.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussMax

Calculates the likelihood probability given multiple observations and a Gaussian mixture component, using the maximum operation.

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussMax_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
   Ipp32s val, int scaleFactor);

IppStatus ippsLogGaussMax_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
   pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
   Ipp32s val, int scaleFactor);

IppStatus ippsLogGaussMax_16s32f_D2(const Ipp16s* pSrc, int step, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height,
   Ipp32f val);

IppStatus ippsLogGaussMax_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s* pMean,
   const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);

IppStatus ippsLogGaussMax_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
   pMean, const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height,
   Ipp32f val);

IppStatus ippsLogGaussMax_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean,
   const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);

IppStatus ippsLogGaussMax_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
   pMean, const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height,
   Ipp64f val);

IppStatus ippsLogGaussMax_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
   const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
```

Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussMax_IdVar_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
   const Ipp16s* pMean, int width, Ipp32s* pSrcDst, int height, Ipp32s val,
   int scaleFactor);

IppStatus ippsLogGaussMax_IdVar_16s32s_D2LSfs(const Ipp16s** mSrc, const
   Ipp16s* pMean, int width, Ipp32s* pSrcDst, int height, Ipp32s val, int
   scaleFactor);
```

```
IppStatus ippsLogGaussMax_IdVar_16s32f_D2(const Ipp16s* pSrc, int step, const
   Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussMax_IdVar_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
   pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussMax_IdVar_32f_D2(const Ipp32f* pSrc, int step, const
   Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussMax_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
   pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussMax_IdVar_64f_D2(const Ipp64f* pSrc, int step, const
   Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussMax_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
   pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
```

Arguments

pSrc	Pointer to the observation vector [*height***step*].
mSrc	Pointer to the observation matrix [*height*][*width*].
step	Row step in the observation vector *pSrc*.
pMean	Pointer to the mean vector [*width*].
pVar	Pointer to the variance vector [*width*].
width	Length of the mean and variance vectors.
pSrcDst	Pointer to the likelihood vector [*height*].
height	Length of the vector *pSrcDst*.
val	Gaussian constant.
scaleFactor	Scale factor for the $V[i]$ values, refer to "Integer Scaling" in Chapter 2.

Discussion

The function `ippsLogGaussMax` calculates the observation probability for multiple observation vectors and accumulates the resulting probabilities in the vector *pSrcDst* using the "maximum" operation.

For functions without the `IdVar` suffix, the covariance matrix is assumed to be inverse diagonal:

For functions with the additional `D2` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \ , \ 0 \le i < height.$$

For functions with the additional `D2L` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \ , \ 0 \le i < height.$$

For functions with the `IdVar` suffix, the covariance matrix is assumed to be identity:

For functions with the additional `D2` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \ , \ 0 \le i < height.$$

For functions with the additional `D2L` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \ , \ 0 \le i < height.$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *mSrc*, *pMean*, *pVar*, or *pSrcDst* pointer is null.

ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussMaxMultiMix

Calculate the likelihood probability for multiple Gaussian mixture components, using the maximum operation.

```
IppStatus ippsLogGaussMaxMultiMix_16s32s_D2Sfs(const Ipp16s* pMean, const
   Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32s* pVal,
   Ipp32s* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_16s32s_D2LSfs(const Ipp16s** mMean, const
   Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32s* pVal, Ipp32s*
   pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_16s32f_D2(const Ipp16s* pMean, const Ipp16s*
   pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f* pVal, Ipp32f*
   pSrcDst, int height);
IppStatus ippsLogGaussMaxMultiMix_16s32f_D2L(const Ipp16s** mMean, const
   Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal, Ipp32f*
   pSrcDst, int height);
IppStatus ippsLogGaussMaxMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f*
   pVar, int step, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f*
   pSrcDst, int height);
IppStatus ippsLogGaussMaxMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
   mVar, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f* pSrcDst,
   int height);
IppStatus ippsLogGaussMaxMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f*
   pVar, int step, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f*
   pSrcDst, int height);
IppStatus ippsLogGaussMaxMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
   mVar, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f* pSrcDst,
   int height);
```

Arguments

pMean	Pointer to the mean vector [*height*step*].
pVar	Pointer to the variance vector [*height*step*].
mMean	Pointer to the mean matrix [*height*][*width*].
mVar	Pointer to the variance matrix [*height*][*width*].
step	Row step in the mean and variance vectors.
width	Number of columns in the mean and variance matrices.
pVal	Pointer to the weight constant vector [*height*].
pSrcDst	Pointer to the likelihood vector [*height*].
height	Number of rows in the mean and variance matrices.
scaleFactor	Scale factor for the $V[i]$ values, refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLogGaussMaxMultiMix calculates the observation probability for multiple Gaussian mixture components and accumulates the resulting probabilities in the vector *pSrcDst* using the "maximum" operation. The covariance matrix is assumed to be inverse diagonal.

For functions with the D2 suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \quad , \quad 0 \le i < height.$$

For functions with the D2L suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \quad , \quad 0 \le i < height.$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mMean*, *mVar*, *pMean*, *pVar*, *pVal*, or *pSrcDst* pointer is null.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussAdd

Calculates the likelihood probability for multiple observation vectors.

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussAdd_16s32f_D2(const Ipp16s* pSrc, int step, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
   Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussAdd_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s* pMean,
   const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
   scaleFactor);
IppStatus ippsLogGaussAdd_32f_D2(const Ipp32f** pSrc, int step, const Ipp32f*
   pMean, const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussAdd_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean,
   const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussAdd_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
   pMean, const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f
   val);
IppStatus ippsLogGaussAdd_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
   const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
```

Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussAdd_IdVar_16s32f_D2(const Ipp16s* pSrc, int step, const
   Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
   scaleFactor);
```

```
IppStatus ippsLogGaussAdd_IdVar_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
   pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
   scaleFactor);
IppStatus ippsLogGaussAdd_IdVar_32f_D2(const Ipp32f* pSrc, int step, const
   Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussAdd_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
   pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussAdd_IdVar_64f_D2(const Ipp64f* pSrc, int step, const
   Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussAdd_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
   pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
```

Arguments

pSrc	Pointer to the observation vector [*height***step*].
mSrc	Pointer to the observation matrix [*height*][*width*].
step	Row step in the observation vector *pSrc*.
pMean	Pointer to the mean vector [*width*].
pVar	Pointer to the variance vector [*width*].
width	Length of the mean and variance vectors.
pSrcDst	Pointer to the likelihood vector [*height*].
height	Number of observation vectors.
val	Weight constant.
scaleFactor	Scale factor for the $V[i]$ values, refer to "Integer Scaling" in Chapter 2.

Discussion

The function `ippsLogGaussAdd` calculates the observation probability for multiple observation vectors and accumulates the resulting probabilities in the vector *pSrcDst*.

For functions without the `IdVar` suffix, the covariance matrix is assumed to be inverse diagonal:

For functions with the additional `D2` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

For functions with the additional `D2L` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

For functions with the `IdVar` suffix, the covariance matrix is assumed to be identity:

For functions with the additional `D2` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

For functions with the additional `D2L` suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *mSrc*, *pMean*, *pVar*, or *pSrcDst* pointer is null.

ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussAddMultiMix

Calculates the likelihood probability for multiple Gaussian mixture components.

```
IppStatus ippsLogGaussAddMultiMix_16s32f_D2(const Ipp16s* pMean, const Ipp16s*
   pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f pVal, Ipp32f*
   pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussAddMultiMix_16s32f_D2L(const Ipp16s** mMean, const
   Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal, Ipp32f*
   pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussAddMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f*
   pVar, int step, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f*
   pSrcDst, int height);
IppStatus ippsLogGaussAddMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
   mVar, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f* pSrcDst,
   int height);
IppStatus ippsLogGaussAddMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f*
   pVar, int step, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f*
   pSrcDst, int height);
IppStatus ippsLogGaussAddMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
   mVar, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f* pSrcDst,
   int height);
```

Arguments

pMean	Pointer to the mean vector [*height*step*].
pVar	Pointer to the variance vector [*height*step*].
mMean	Pointer to the mean matrix [*height*][*width*].
mVar	Pointer to the variance matrix [*height*][*width*].
step	Row step in mean and variance vectors (in *pMean* elements).

width	Length of the mean and variance matrices.
pVal	Pointer to the weight constant vector [*height*].
pSrcDst	Pointer to the likelihood vector [*height*].
height	Number of Gaussian mixture components.
scaleFactor	Scale factor for the *V*[*i*] values, refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLogGaussAddMultiMix calculates the observation probability for multiple Gaussian mixture components and accumulates the resulting probabilities in the vector *pSrcDst*. The covariance matrix is assumed to be inverse diagonal.
For functions with the D2 suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

For functions with the D2L suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mMean*, *mVar*, *pMean*, *pVar*, *pVal*, or *pSrcDst* pointer is null.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

Model Estimation

This section describes functions that are needed to estimate the parameters of the acoustic and language models.

MeanColumn

Computes the mean values for the column elements.

```
IppStatus ippsMeanColumn_16s_D2(const Ipp16s* pSrc, int height, int step,
   Ipp16s* pDstMean, int width);
IppStatus ippsMeanColumn_16s_D2L(const Ipp16s** mSrc, int height, Ipp16s*
   pDstMean, int width);
IppStatus ippsMeanColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
   ipp32f* pDstMean, int width);
IppStatus ippsMeanColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
   pDstMean, int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
height	Number of rows in the input matrix *mSrc*.
step	Row step in the input vector (measured in *pSrc* elements).
pDstMean	Pointer to the output mean vector [*width*].
width	Number of columns in the input matrix *mSrc*, and also the length of the output mean vector *pDstMean*.

Discussion

The function ippsMeanColumn calculates the mean values for the column elements of the input matrix *mSrc* as follows:

For functions with the D2 suffix,

$$pDstMean[j] = \frac{1}{height} \sum_{i=1}^{height-1} pSrc[i \cdot step + j], \quad 0 \le j < width$$

For functions with the D2L suffix,

$$pDstMean[j] = \frac{1}{height} \sum_{i=1}^{height-1} mSrc[i][j], \quad 0 \le j < width$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, or *pDstMean* pointer is null.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

VarColumn

Calculates the variances for the column elements.

```
IppStatus ippsVarColumn_16s_D2Sfs(const Ipp16s* pSrc, int height, int step,
   Ipp16s* pSrcMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsVarColumn_16s_D2LSfs(const Ipp16s** mSrc, int height, Ipp16s*
   pSrcMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsVarColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
   Ipp32f* pSrcMean, Ipp32f* pDstVar, int width);
IppStatus ippsVarColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
   pSrcMean, Ipp32f* pDstVar, int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].

mSrc	Pointer to the input matrix [*height*][*width*].
height	Number of rows in the input matrix *mSrc*.
step	Row step in the input vector *pSrc*.
pSrcMean	Pointer to the input mean vector [*width*].
pDstVar	Pointer to the output variance vector [*width*].
width	Number of columns in the input matrix *mSrc*, and also the length of the input mean vector *pSrcMean* and the output variance vector *pDstVar*.
scaleFactor	Scale factor, refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsVarColumn calculates the variances for the column elements of the matrix *mSrc* as follows:

For functions with the D2 suffix,

$$pDstVar[j] = \frac{-height \cdot (pSrcMean[j])^2 + \sum_{i=1}^{height-1} pSrc[i \cdot step + j]^2}{height - 1},$$

$0 \le j < width$.

For functions with the D2L suffix

$$pDstVar[j] = \frac{-height \cdot (pSrcMean[j])^2 + \sum_{i=1}^{height-1} mSrc[i][j]^2}{height - 1},$$

$0 \le j < width$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pSrcMean*, or *pDstVar* pointer is null.
ippStsSizeErr	Indicates an error when *width* is less than or equal to 0 or *height* is less than or equal to 1.

ippStsStrideErr	Indicates an error when *step* is less than *width*.

MeanVarColumn

Calculates the means and variances for the column elements of a matrix.

```
IppStatus ippsMeanVarColumn_16s_D2Sfs(const Ipp16s* pSrc, int height, int
   step, Ipp16s* pDstMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_16s_D2LSfs(const Ipp16s** mSrc, int height,
   Ipp16s* pDstMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
   Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsMeanVarColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
   pDstMean, Ipp32f* pDstVar, int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
height	Number of rows in the input matrix *mSrc*.
step	Row step in the input vector *pSrc*.
pDstMean	Pointer to the output mean vector [*width*].
pDstVar	Pointer to the output variance vector [*width*].
width	Number of columns in the input matrix *mSrc*, and also the length of the output mean vector *pDstMean* and variance vector *pDstVar*.
scaleFactor	Scale factor, refer to "Integer Scaling" in Chapter 2. *scaleFactor* argument (and integer scaling) is used for *pDstVar* only, while *pDstMean* elements are not scaled.

Discussion

The function ippsMeanVarColumn calculates both the means and the variances for the column elements of the matrix *mSrc*. See functions ippsMeanColumn and ippsVarColumn for calculation details.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *mSrc*, *pDstMean*, or *pDstVar* pointer is null.
ippStsSizeErr	Indicates an error when *width* is less than or equal to 0 or *height* is less than or equal to 1.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

NormalizeColumn

Normalizes the matrix columns given the column means and variances.

```
IppStatus ippsNormalizeColumn_16s_D2Sfs(Ipp16s* pSrcDst, int step, int height,
   const Ipp16s* pMean, const Ipp16s* pVar, int width, int scaleFactor);
IppStatus ippsNormalizeColumn_16s_D2LSfs(Ipp16s** mSrcDst, int height, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, int scaleFactor);
IppStatus ippsNormalizeColumn_32f_D2(Ipp32f* pSrcDst, int step, int height,
   const Ipp32f* pMean, const Ipp32f* pVar, int width);
IppStatus ippsNormalizeColumn_32f_D2L(Ipp32f** mSrcDst, int height, const
   Ipp32f* pMean, const Ipp32f* pVar, int width);
```

Arguments

pSrcDst	Pointer to the input and output vector [*height***step*].
mSrcDst	Pointer to the input and output matrix [*height*][*width*].
pMean	Pointer to the column mean vector [*width*].

pVar	Pointer to the column variance vector [*width*].
width	Length of the mean and variance vectors.
step	Row step in the input and output vector *pSrcDst*.
height	Number of rows in the input and output matrix *mSrcDst*.

Discussion

The function ippsNormalizeColumn normalizes the columns of the matrix *mSrcDst* as follows:

For functions with the D2 suffix,

$$pSrcDst[i \cdot step + j] = (pSrcDst[i \cdot step + j] - pMean[j]) \cdot pVar[j] ,$$

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

For functions with the D2L suffix,

$$mSrcDst[i][j] = (mSrcDst[i][j] - pMean[j]) \cdot pVar[j] ,$$

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

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcDst*, *mSrcDst*, *pMean*, or *pVar* pointer is null.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

MeanVarAcc

Accumulate the estimates for the mean and variance re-estimation.

```
IppStatus ippsMeanVarAcc_32f(Ipp32f const* pSrc, Ipp32f const* pSrcMean,
   Ipp32f* pDstMeanAcc, Ipp32f* pDstVarAcc, int len, Ipp32f val);
IppStatus ippsMeanVarAcc_64f(Ipp64f const* pSrc, Ipp64f const* pSrcMean,
   Ipp64f* pDstMeanAcc, Ipp64f* pDstVarAcc, int len, Ipp64f val);
```

Arguments

pSrc	Pointer to the observation vector [*len*].
pSrcMean	Pointer to the old mean vector [*len*].
pDstMeanAcc	Pointer to the mean accumulator [*len*].
pDstVarAcc	Pointer to the variance accumulator [*len*].
len	Length of the observation vector.
val	Constant value in the re-estimation.

Discussion

The function ippsMeanVarAcc accumulates the estimates for the mean and variance re-estimation in the forward-backward algorithm. The calculation is as follows:

$$pDstMeanAcc[i] = pDstMeanAcc[i] + val \cdot (pSrc[i] - pSrcMean[i]) ,$$

$$pDstVarAcc[i] = pDstVarAcc[i] + val \cdot (pSrc[i] - pSrcMean[i])^2 ,$$

$$0 \le i < len .$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pSrcMean*, *pDstMeanAcc*, or *pDstVarAcc* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

GaussianDist

Calculates the distance between two Gaussians.

```
IppStatus ippsGaussianDist_32f(const Ipp32f* pMean1, const Ipp32f* pVar1,
   const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult, Ipp32f
   wgt1, Ipp32f det1, Ipp32f wgt2, Ipp32f det2);
IppStatus ippsGaussianDist_64f(const Ipp64f* pMean1, const Ipp64f* pVar1,
   const Ipp64f* pMean2, const Ipp64f* pVar2, int len, Ipp64f* pResult, Ipp64f
   wgt1, Ipp64f det1, Ipp64f wgt2, Ipp64f det2);
```

Arguments

pMean1	Pointer to the mean vector of the first Gaussian [*len*].
pVar1	Pointer to the variance vector of the first Gaussian [*len*].
pMean2	Pointer to the mean vector of the second Gaussian [*len*].
pVar2	Pointer to the variance vector of the second Gaussian [*len*].
len	Length of the mean and variance vectors.
pResult	Pointer to the distance value.
wgt1	Weight of the first Gaussian.
det1	Determinant of the first Gaussian (in the logarithmic representation).
wgt2	Weight of the second Gaussian.
det2	Determinant of the second Gaussian (in the logarithmic representation).

Discussion

The function `ippsGaussianDist` calculates the distance between two Gaussians as follows:

$$pResult[0] = (wgt1 \cdot det1) + (wgt2 \cdot det2) - (wgt1 + wgt2) \cdot (len \cdot \ln(2\pi) - \ln(V)),$$

where

$$V = \prod_{i=0}^{len-1} \frac{W_1[i] + W_2[i] - (wgt1 \cdot pMean1[i] + wgt2 \cdot pMean2[i])^2}{wgt1 + wgt2},$$

and

$$W_1[i] = wgt1 \cdot (pVar1[i] + pMean1[i]^2),$$

$$W_2[i] = wgt2 \cdot (pVar2[i] + pMean2[i]^2)$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMean1*, *pMean2*, *pVar1*, *pVar2*, or *pResult* pointer is null.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

GaussianSplit

Splits a single Gaussian component into two with the same variance.

```
IppStatus ippsGaussianSplit_32f(Ipp32f* pMean1, Ipp32f* pVar1, Ipp32f* pMean2,
   Ipp32f* pVar2, int len, Ipp32f val);
IppStatus ippsGaussianSplit_64f(Ipp64f* pMean1, Ipp64f* pVar1, Ipp64f* pMean2,
   Ipp64f* pVar2, int len, Ipp64f val);
```

Arguments

pMean1	Pointer to the input Gaussian mean vector, also the mean vector of the first Gaussian after splitting [*len*].
pVar1	Pointer to the input Gaussian variance vector, also the variance vector of the first Gaussian after splitting [*len*].
pMean2	Pointer to the mean vector of the second Gaussian after splitting [*len*].

pVar2	Pointer to the variance vector of the second Gaussian after splitting [*len*].
len	Length of the mean and variance vectors.
val	Variance perturbation value.

Discussion

The function ippsGaussianSplit splits the Gaussian component into two Gaussian mixture components as follows:

$pMean1[i] = pMean1[i] + val \cdot \sqrt{pVar1[i]}$,

$pMean2[i] = pMean1[i] - val \cdot \sqrt{pVar1[i]}$,

$pVar2[i] = pVar1[i]$,

$0 \le i < len$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMean1*, *pMean2*, *pVar1*, or *pVar2* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

GaussianMerge

Merges two Gaussian probability distribution functions.

```
IppStatus ippsGaussianMerge_32f(const Ipp32f* pMean1, const Ipp32f* pVar1,
   const Ipp32f* pMean2, const Ipp32f* pVar2, Ipp32f* pDstMean, Ipp32f*
   pDstVar, int len, Ipp32f* pDstDet, Ipp32f wgt1, Ipp32f wgt2);
IppStatus ippsGaussianMerge_64f(const Ipp64f* pMean1, const Ipp64f* pVar1,
   const Ipp64f* pMean2, const Ipp64f* pVar2, Ipp64f* pDstMean, Ipp64f*
   pDstVar, int len, Ipp64f* pDstDet, Ipp64f wgt1, Ipp64f wgt2);
```

Arguments

pMean1	Pointer to the mean vector of the first Gaussian [*len*].
pVar1	Pointer to the variance vector of the first Gaussian [*len*].
pMean2	Pointer to the mean vector of the second Gaussian [*len*].
pVar2	Pointer to the variance vector of the second Gaussian [*len*].
len	Length of the mean and variance vectors.
pDstMean	Pointer to the mean vector of the merged Gaussian [*len*].
pDstVar	Pointer to the variance vector of the merged Gaussian [*len*].
pDstDet	Pointer to the determinant of the merged Gaussian.
wgt1	Weight of the first Gaussian.
wgt2	Weight of the second Gaussian.

Discussion

The function ippsGaussianMerge merges two Gaussian probability distribution functions as follows:

$$pDstMean[i] = \frac{wgt1 \cdot pMean1[i] + wgt2 \cdot pMean2[i]}{wgt1 + wgt2},$$

$$pDstVar[i] = \frac{wgt1^2 \cdot (pVar1[i] + pMean1[i]^2) + wgt2^2 \cdot (pVar2[i] + pMean2[i]^2)}{wgt1 + wgt2} - pDstMean[i]^2,$$

$$pDstDet[0] = len \cdot \ln(2\pi) - \sum_{i=0}^{len-1} \ln pDstVar[i],$$

$0 \le i < len$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMean1*, *pMean2*, *pVar1*, *pVar2*, *pDstMean*, *pDstVar*, or *pDstDet* pointer is null.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

BhatDist

Calculates the Bhattacharia distance between two Gaussians.

```
IppStatus ippsBhatDist_32f(const Ipp32f* pMean1, const Ipp32f* pVar1, const
   Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult);
IppStatus ippsBhatDist_32f64f(const Ipp32f* pMean1, const Ipp32f* pVar1, const
   Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp64f* pResult);
IppStatus ippsBhatDistSLog_32f(const Ipp32f* pMean1, const Ipp32f* pVar1,
   const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult, Ipp32f
   sumLog1, Ipp32f sumLog2);
IppStatus ippsBhatDistSLog_32f64f(const Ipp32f* pMean1, const Ipp32f* pVar1,
   const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp64f* pResult, Ipp32f
   sumLog1, Ipp32f sumLog2);
```

Arguments

pMean1	Pointer to the first mean vector [*len*].
pVar1	Pointer to the first variance vector [*len*].
pMean2	Pointer to the second mean vector [*len*].
pVar2	Pointer to the second variance vector [*len*].
pResult	Pointer to the result.
len	Length of the input mean and variance vectors.
sumLog1	Sum of the first Gaussian variance in the logarithmic representation.
sumLog2	Sum of the second Gaussian variance in the logarithmic representation.

Discussion

The function ippsBhatDist calculates the Bhattacharia distance between two Gaussians as follows:

$$pResult[0] = \frac{1}{4}\sum_{i=0}^{len-1}\frac{(pMean1[i]-pMean2[i])^2}{pVar1[i]+pVar2[i]} + \frac{1}{2}\sum_{i=0}^{len-1}\left(\ln\left(\frac{pVar1[i]+pVar2[i]}{2}\right)-\frac{\ln(pVar1[i])+\ln(pVar2[i])}{2}\right)$$

The function ippsBhatDistSLog calculates the Bhattacharia distance between two Gaussians as follows:

$$pResult[0] = \frac{1}{4}\sum_{i=0}^{len-1}\frac{(pMean1[i]-pMean2[i])^2}{pVar1[i]+pVar2[i]} + \frac{1}{2}\sum_{i=0}^{len-1}\ln\left(\frac{pVar1[i]+pVar2[i]}{2}\right)-\frac{sumLog1+sumLog2}{4}$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pMean1*, *pVar1*, *pMean2*, *pVar2*, or *pResult* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsLnZeroArg	Indicates a warning that a zero value was detected in the input vector. The execution is not aborted. The result value is set to –Inf if there is no negative element in the vector.
ippStsLnNegArg	Indicates a warning that negative values were detected in the input vector. The execution is not aborted. The result value is set to NaN.

UpdateMean

Updates the mean vector in the EM training algorithm.

```
IppStatus ippsUpdateMean_32f(const Ipp32f* pMeanAcc, Ipp32f* pMean, int len,
   Ipp32f meanOcc);
IppStatus ippsUpdateMean_64f(const Ipp64f* pMeanAcc, Ipp64f* pMean, int len,
   Ipp64f meanOcc);
```

Arguments

pMeanAcc	Pointer to the mean accumulator [*len*].
pMean	Pointer to the mean vector [*len*].
len	Length of the mean vector.
meanOcc	Occupation sum of the Gaussian mixture.

Discussion

The function ippsUpdateMean calculates the updated mean vector in the EM (Expectation-Maximization) training algorithm as follows::

$$pMean[i] = pMean[i] + \frac{pMeanAcc[i]}{meanOcc}, 0 \le i < len.$$

Note that if $meanOcc \le 0$, the mean vector *pMean* is not updated.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pMean* or *pMeanAcc* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsZeroOcc	Indicates a warning when *meanOcc* is equal to 0.
ippStsNegOcc	Indicates an error when *meanOcc* is less than 0.

UpdateVar

Updates the variance vector in the EM training algorithm.

```
IppStatus ippsUpdateVar_32f(const Ipp32f* pMeanAcc, const Ipp32f* pVarAcc,
   const Ipp32f* pVarFloor, Ipp32f* pVar, int len, Ipp32f meanOcc, Ipp32f
   varOcc);
IppStatus ippsUpdateVar_64f(const Ipp64f* pMeanAcc, const Ipp64f* pVarAcc,
   const Ipp64f* pVarFloor, Ipp64f* pVar, int len, Ipp64f meanOcc, Ipp64f
   varOcc);
```

Arguments

pMeanAcc	Pointer to the mean accumulator [*len*].
pVarAcc	Pointer to the variance accumulator [*len*].
pVarFloor	Pointer to the variance floor vector [*len*].
pVar	Pointer to the variance vector [*len*].
len	Length of the variance.
meanOcc	Occupation sum of the Gaussian mixture.
varOcc	Square occupation sum of the variance mixture.

Discussion

The function `ippsUpdateVar` calculates the updated variance vector in the EM algorithm. The covariance matrix is assumed to be diagonal. The accumulators are calculated from the training data. The update equation is as follows:

$$pVar[i] = \max\left[varFloor[i],\ \frac{varAcc[i]}{varOcc} - \left(\frac{meanAcc[i]}{meanOcc}\right)^2\right],\ 0 \le i < len.$$

Note that if $meanOcc \le 0$ or $varOcc \le 0$, the variance vector *pVar* is not updated.

Return Value

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pMeanAcc*, *pVarAcc*, *pVarFloor*, or *pVar* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsZeroOcc	Indicates a warning when *meanOcc* or *varOcc* is equal to 0.
ippStsNegOcc	Indicates an error when *meanOcc* and *varOcc* are less than 0.
ippStsResFloor	Indicates a warning when all variances are floored.

UpdateWeight

Updates the weight values of Gaussian mixtures in the EM training algorithm.

```
IppStatus ippsUpdateWeight_32f(const Ipp32f* pWgtAcc, Ipp32f* pWgt, int len,
   Ipp32f* pWgtSum, Ipp32f wgtOcc, Ipp32f wgtThresh);
IppStatus ippsUpdateWeight_64f(const Ipp64f* pWgtAcc, Ipp64f* pWgt, int len,
   Ipp64f* pWgtSum, Ipp64f wgtOcc, Ipp64f wgtThresh);
```

Arguments

pWgtAcc	Pointer to the weight accumulator [*len*].
pWgt	Pointer to the weight vector [*len*].
len	Number of Gaussian mixture components.
pWgtSum	Pointer to the output sum of weight values.
wgtOcc	Nominator of the weight update equation.
wgtThresh	Threshold for the weight values.

Discussion

The function ippsUpdateWeight calculates the updated weight values for a Gaussian mixture. The accumulators are calculated from the training data. The update equation is as follows:

$$pWgt[i] = \max\left(\frac{pWgtAcc[i]}{wgtOcc},\ wgtThresh\right),\ 0 \le i < len$$

$$pWgtSum[0] = \sum_{i=0}^{len-1} pWgt[i] .$$

Note that if $wgtOcc \leq 0$, the weight vector *pWgt* is not updated.

This function can also be used to update the HMM transition matrix.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pWgt*, *pWgtAcc*, or *pWgtSum* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsZeroOcc	Indicates a warning when *wgtOcc* is equal to 0.
ippStsNegOcc	Indicates an error when *wgtOcc* is less than 0.
ippStsResFloor	Indicates a warning when all weights are floored.

UpdateGConst

Updates the fixed constant in the Gaussian output probability density function.

```
IppStatus ippsUpdateGConst_32f(const Ipp32f* pVar, int len, Ipp32f* pDet);
IppStatus ippsUpdateGConst_64f(const Ipp64f* pVar, int len, Ipp64f* pDet);
IppStatus ippsUpdateGConst_DirectVar_32f(const Ipp32f* pVar, int len,
   Ipp32f* pDet);
IppStatus ippsUpdateGConst_DirectVar_64f(const Ipp64f* pVar, int len,
   Ipp64f* pDet);
```

Arguments

pVar	Pointer to the variance vector [*len*].
len	Dimension of the variance vector.

pDet	Pointer to the result value.

Discussion

This function calculates the fixed variance constant.
For functions without the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be inverse diagonal:

$$pDet[0] = len \cdot \ln 2\pi - \sum_{i=o}^{len-1} \ln(pVar[i])$$

For functions with the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be diagonal:

$$pDet[0] = len \cdot \ln 2\pi + \sum_{i=o}^{len-1} \ln(pVar[i])$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pVar* or *pDet* pointer is null.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsLnZeroArg`	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The result value is set to `–Inf` if there are no negative elements in the vector.
`ippStsLnNegArg`	Indicates a warning for negative input vector elements. Operation execution is not aborted. The result value is set to `NaN`.

OutProbPreCalc

Pre-calculates the part of Gaussian mixture output probability that is irrelevant to observation vectors.

```
IppStatus ippsOutProbPreCalc_32f(const Ipp32f* pWeight, const Ipp32f* pSrc,
   Ipp32f* pDst, int len);
IppStatus ippsOutProbPreCalc_32f_I(const Ipp32f* pWeight, Ipp32f* pSrcDst,
   int len);
IppStatus ippsOutProbPreCalc_64f(const Ipp64f* pWeight, const Ipp64f* pSrc,
   Ipp64f* pDst, int len);
IppStatus ippsOutProbPreCalc_64f_I(const Ipp64f* pWeight, Ipp64f* pSrcDst,
   int len);
```

Arguments

pWeight	Pointer to the Gaussian mixture weight vector [*len*].
pSrc	Pointer to the input vector calculated from ippsUpdateGConst function.
pSrcDst	Pointer to the input and output vector calculated from ippsUpdateGConst function.
pDst	Pointer to the resulting vector [*len*].
len	Number of mixtures in the HMM state.

Discussion

This function pre-calculates the part of the Gaussian mixture output probability that is irrelevant to the observation vectors.

For the function ippsOutProbPreCalc,

$pDst[i] = pWeight[i] - 0.5 \cdot pSrc[i]$, $0 \le i < len$.

For the function ippsOutProbPreCalc_I,

$pSrcDst[i] = pWeight[i] - 0.5 \cdot pSrcDst[i]$, $0 \le i < len$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pWeight*, *pDet*, or *pVal* pointer is null.
ippStsSizeErr	Indicates an error when the *len* is less than or equal to zero.

DcsClustLAccumulate

Updates the accumulators for calculating the state-cluster likelihood in the decision-tree clustering algorithm.

```
IppStatus ippsDcsClustLAccumulate_DirectVar_32f(const Ipp32f* pMean, const
   Ipp32f* pVar, Ipp32f* pDstSum, Ipp32f* pDstSqr, int len, Ipp32f occ);
IppStatus ippsDcsClustLAccumulate_DirectVar_64f(const Ipp64f* pMean, const
   Ipp64f* pVar, Ipp64f* pDstSum, Ipp64f* pDstSqr, int len, Ipp64f occ);
IppStatus ippsDcsClustLAccumulate_32f(const Ipp32f* pMean, const Ipp32f* pVar,
   Ipp32f* pDstSum, Ipp32f* pDstSqr, int len, Ipp32f occ);
IppStatus ippsDcsClustLAccumulate_64f(const Ipp64f* pMean, const Ipp64f* pVar,
   Ipp64f* pDstSum, Ipp64f* pDstSqr, int len, Ipp64f occ);
```

Arguments

pMean	Pointer to the mean vector of an HMM state in the cluster [*len*].
pVar	Pointer to the variance vector of an HMM state in the cluster[*len*].
pDstSum	Pointer to the summation part of the accumulator[*len*].
pDstSqr	Pointer to the square sum part of the accumulator[*len*].
len	Length of the mean and variance vectors.
occ	Occupation counts of the HMM state

Discussion

This function updates the accumulators in the decision-tree clustering algorithm. The accumulators are used to calculate the likelihood of an HMM state cluster.

The accumulation equations are as follows:

For functions without the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be inverse diagonal, and

$pDstSum[i] = pMean[i] \cdot occ$,

$pDstSqr[i] = \left[\frac{1}{pVar[i]} + (pMean[i])^2\right] \cdot occ$, for $0 \le i < len$.

For functions with the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be diagonal, and

$pDstSum[i] = pMean[i] \cdot occ$,

$pDstSqr[i] = [pVar[i] + (pMean[i])^2] \cdot occ$, for $0 \le i < len$.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pMean*, *pVar*, *pDstSum*, or *pDstSqr* pointer is null.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

DcsClustLCompute

Calculates the likelihood of an HMM state cluster in the decision-tree state-clustering algorithm.

```
IppStatus ippsDcsClustLCompute_32f64f(const Ipp32f* pSrcSum, const Ipp32f*
   pSrcSqr, int len, Ipp64f* pDst, Ipp32f occ);
IppStatus ippsDcsClustLCompute_64f(const Ipp64f* pSrcSum, const Ipp64f*
   pSrcSqr, int len, Ipp64f* pDst, Ipp64f occ);
```

Arguments

pSrcSum	Pointer to the summation part of the accumulator [*len*].
pSrcSqr	Pointer to the square sum part of the accumulator [*len*].
pDst	Pointer to the resulting likelihood value.
occ	Occupation sum of the HMM state cluster.

Discussion

The function ippsDcsClustLCompute calculates the likelihood of an HMM state cluster, according to the accumulators computed by ippsDcsClustLAccumulate function. The likelihood value is used to determine the splitting of a decision tree node in the decision-tree state-clustering algorithm. The calculation is as follows:

$$pDst[0] = -\frac{1}{2}occ \cdot \left\{ len \cdot [1 + \ln 2\pi - 2\ln(occ)] + \right.$$
$$\left. + \sum_{i=0}^{len-1} \ln[pSrcSqr[i] \cdot occ - (pSrcSum[i])^2] \right\}$$

Note that if $occ = 0$, $pDst[0]$ is set to IPPLOGZERO.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcSum*, *pSrcSqr* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0 or *occ* is less than or equal to 0.
ippStsZeroOcc	Indicates a warning when *occ* is equal to 0.
ippStsNegOcc	Indicates an error when *occ* is less than 0.
ippStsLnZeroArg	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The result value is set to –Inf if there are no negative elements in the vector.
ippStsLnNegArg	Indicates a warning for negative input vector elements. Operation execution is not aborted. The result value is set to NaN.

Model Adaptation

This section describes functions that can be used to adapt the acoustic and language models. The adaptation algorithms adjust the parameters of existing models to match the characteristics set by the users, given a few learning samples.

AddMulColumn

Adds a weighted matrix column to the other column.

```
IppStatus ippsAddMulColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height, int
   col1, int col2, int row1, const Ipp64f val);
```

Arguments

mSrcDst	Pointer to the source and destination matrix [*height*][*width*].
width	Number of columns in the matrix *mSrcDst*.
height	Number of rows in the matrix *mSrcDst*.
col1	Column number of the first operand.
col2	Column number of the second operand.
row1	Starting row number.
val	Weight factor.

Discussion

The function AddMulColumn adds a matrix column weighted by *val* to the other matrix column.The function is used for fast execution of the SVD algorithm.
The calculation is as follows:

$$mSrcDst[i][col2] = mSrcDst[i][col2] + mSrcDst[i][col1] \cdot val,$$

for $row1 \le i < height.$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *mSrcDst* pointer is null.
ippStsSizeErr	Indicates an error when *height*, *width*, *col1*, *col2*, or *row1* is less than or equal to 0; or *col1* or *col2* is greater than or equal to *width*; or *row1* is greater than or equal to *height*.

AddMulRow

Adds a weighted vector to the other vector.

```
ippsAddMulRow_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len,
   const Ipp64f val);
```

pSrc	Pointer to the source vector [*len*].
pSrcDst	Pointer to the source and destination vector [*len*].
len	Length of the source and destination vectors.
val	Weight factor.

Discussion

The function AddMulRow adds a vector weighted by *val* to the other vector. The function is used for fast execution of the SVD algorithm.The calculation is as follows:

$pSrcDst[i] = pSrcDst[i] + pSrc[i] \cdot val, \quad 0 \le i < len$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pSrcDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

QRTransColumn

Performs the QR transformation.

```
IppStatus ippsQRTransColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height,
    int col1, int col2, const Ipp64f val1, const Ipp64f val2);
```

Arguments

mSrcDst	Pointer to the source matrix and destination matrix [*height*][*width*].
width	Number of columns in the matrix *mSrcDst*.
height	Number of rows in the matrix *mSrcDst*.
col1	First column number.
col2	Second column number.
val1	First weight factor.
val2	Second weight factor.

Discussion

The `ippsQRTransColumn` performs the *QR* transform. The function is used for fast execution of the SVD algorithm. The calculation is as follows:

$$mSrcDst[i][col2] = mSrcDst[i][col2] \cdot val1 + mSrcDst[i][col1] \cdot val2 ,$$

$$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val1 + mSrcDst[i][col2] \cdot val2 ,$$

for $0 \le i < height$.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *mSrcDst* pointer is null.

ippStsSizeErr	Indicates an error when *height*, *width*, *col1*, or *col2* is less than or equal to 0; or *col1* or *col2* is greater than or equal to *width*; or *col2* is greater than or equal to *height*.

DotProdColumn

Calculates the dot product of two matrix columns.

```
IppStatus ippsDotProdColumn_64f_D2L(const Ipp64f** mSrc, int width,
   int height, Ipp64f* pSum, int col1, int col2, int row1);
```

Arguments

mSrc	Pointer to the source matrix [*height*][*width*].
width	Number of columns in the matrix *mSrc*.
height	Number of rows in the matrix *mSrc*.
pSum	Pointer to the computed sum.
col1	First column number.
col2	Second column number.
row1	First row number.

Discussion

The function ippsDotProdColumn calculates the dot product of two matrix columns. The function is used for fast execution of the SVD algorithm. The calculation is as follows:

$$pSum[0] = \sum_{i=row1}^{height-1} mSrc[i][col1] \cdot mSrc[i][col2]$$

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when *mSrc* or *pSum* pointer is null.
ippStsSizeErr	Indicates an error when *height*, *width*, *col1*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *col1* is greater than or equal to *width*.

MulColumn

Multiplies a matrix column by a value.

```
IppStatus ippsMulColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height,
   int col1, int row1, const Ipp64f val);
```

Arguments

mSrcDst	Pointer to the source and destination matrix [*height*][*width*].
width	Number of columns in the matrix *mSrcDst*.
height	Number of rows in the matrix *mSrcDst*.
col1	First column number.
row1	First row number.
val	Weight factor.

Discussion

The function ippsMulColumn multiplies a column of the *mSrcDst* matrix by *val*. The function is used for fast execution of the SVD algorithm. The calculation is as follows:

$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val, \quad row1 \le i < height$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *mSrcDst* pointer is null.

ippStsSizeErr	Indicates an error when *height*, *width*, *col1*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *col1* is greater than or equal to *width*.

SumColumnAbs

Calculates the absolute sum of matrix column elements.

```
IppStatus ippsSumColumnAbs_64f_D2L(const Ipp64f** mSrc, int width,
   int height, Ipp64f* pSum, int col1, int row1);
```

Arguments

mSrc	Pointer to the source matrix [*height*][*width*].
width	Number of columns in the matrix *mSrc*.
height	Number of rows in the matrix *mSrc*.
pSum	Pointer to the computed sum.
col1	First column number.
row1	First row number.

Discussion

The function ippsSumColumnAbs calculates the absolute sum of the matrix column elements. The function is used for fast execution of the SVD algorithm.
The calculation is as follows:

$$pSum[0] = \sum_{i=row1}^{height-1} |mSrc[i][col1]|$$

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when *mSrc* or *pSum* pointer is null.
ippStsSizeErr	Indicates an error when *width*, *height*, *col1*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *col1* is greater than or equal to *width*.

SumColumnSqr

Calculates the square sums of weighted matrix column elements.

```
IppStatus ippsSumColumnSqr_64f_D2L(Ipp64f** mSrcDst, int width, int height,
   Ipp64f* pSum, int col1, int row1, const Ipp64f val);
```

Arguments

mSrcDst	Pointer to the source and destination matrix [*height*][*width*].
width	Number of columns in the matrix *mSrcDst*.
height	Number of rows in the matrix *mSrcDst*.
pSum	Pointer to the value of the computed sum.
col1	First column number.
row1	Second row number.
val	Weight factor.

Discussion

The function ippsSumColumnSqr multiplies the columns of the matrix *mSrcDst* by *val* and calculates the square sums of the column elements. This function is used for fast execution of the SVD algorithm. The calculation is as follows:

$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val, \quad 0 \le i < height$

$$pSum[0] = \sum_{i=row1}^{height-1} (mSrcDst[i][col1])^2$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *mSrcDst* or *pSum* pointer is null.
ippStsSizeErr	Indicates an error when *width*, *height*, *col1*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *col1* is greater than or equal to *width*.

SumRowAbs

Calculates the absolute sum of the vector elements.

```
IppStatus ippsSumRowAbs_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
```

Arguments

pSrc	Pointer to the source vector [*len*].
pSum	Pointer to the value of the computed sum.
len	Length of the source vector *pSrc*.

Discussion

The function ippsSumRowAbs calculates the absolute sum of the vector elements as follows:

$$pSum[0] = \sum_{i=0}^{len-1} |pSrc[i]|$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pSum* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

SumRowSqr

Calculates the square sum of weighted vector elements.

```
IppStatus ippsSumRowSqr_64f(Ipp64f* pSrcDst, int len, Ipp64f* pSum,
   const Ipp64f val);
```

Arguments

pSrcDst	Pointer to the source and destination vector [*len*].
len	Length of the source vector *pSrcDst*.
pSum	Pointer to the value of the computed sum.
val	Weight factor.

Discussion

The function ippsSumRowSqr multiplies the vector *pSrcDst* by *val* and calculates the square sum of the vector elements as follows:

$pSrcDst[i] = pSrcDst[i] \cdot val, \quad 0 \le i < len$

$$pSum[0] = \sum_{i=0}^{len-1} (pSrcDst[i])^2.$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcDst* or *pSum* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

SVD

Performs Single Value Decomposition on a matrix.

```
IppStatus ippsSVD_64f_D2(const Ipp64f* pSrcA, Ipp64f* pDstU, int height,
   Ipp64f* pDstW, Ipp64f* pDstV, int width, int step, int nIter);
IppStatus ippsSVD_64f_D2_I(Ipp64f* pSrcDstA, int height, Ipp64f* pDstW,
   Ipp64f* pDstV, int width, int step, int nIter);
IppStatus ippsSVD_64f_D2L(const Ipp64f** mSrcA, Ipp64f** mDstU, int height,
   Ipp64f* pDstW, Ipp64f** mDstV, int width, int nIter);
IppStatus ippsSVD_64f_D2L_I(Ipp64f** mSrcDstA, int height, Ipp64f* pDstW,
   Ipp64f** mDstV, int width, int nIter);
```

Arguments

pSrcA	Pointer to the input vector A [*height*step*].
pDstU	Pointer to the output vector U [*height*step*].
pSrcDstA	Pointer to the input matrix A and output matrix U [*height*step*].
pDstV	Pointer to the output vector V [*width*step*].
mSrcA	Pointer to the input matrix A [*height*][*width*].
mDstU	Pointer to the output matrix U [*height*][*width*].
mSrcDstA	Pointer to the input matrix A and output matrix U [*height*][*width*].
pDstW	Pointer to the output vector W [*width*].
mDstV	Pointer to the output matrix V [*width*][*width*].
height	Number of rows in the input matrix.
width	Number of columns in the input matrix.
step	Row step in the vector *pSrcA*, *pSrcDstA*, or *pDstV*.
nIter	Number of iterations for diagonalization.

Discussion

The function `ippsSVD` performs Single Value Decomposition (SVD) on the input matrix A. The output matrices U, W, and V meet the following condition:

$$A = U \circ W \circ V^T,$$

where the matrix U is column-orthogonal, the matrix W is diagonal (stored as a vector), and the matrix V is orthogonal. V^T is the transpose of the matrix V.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is null.
`ippStsSizeErr`	Indicates an error when *`height`*, *`width`*, *`step`* or *`nIter`* is less than or equal to 0, or *`width`* is greater than *`step`*.
`ippStsSVDCnvg`	Indicates an error when the SVD algorithm has not converged after *`nIter`* iterations.

WeightedSum

Calculates the weighted sums of two input vector elements.

```
IppStatus ippsWeightedSum_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
   Ipp16s* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSumHalf_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
   Ipp16s* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSum_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
   Ipp32f* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSumHalf_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
   Ipp32f* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSum_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
   Ipp64f* pDst, int len, Ipp64f weight1, Ipp64f weight2);
IppStatus ippsWeightedSumHalf_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
   Ipp64f* pDst, int len, Ipp64f weight1, Ipp64f weight2);
```

Arguments

pSrc1	Pointer to the first input vector [*len*].
pSrc2	Pointer to the second input vector [*len*].
pDst	Pointer to the output vector [*len*].
len	Length of the input and output vectors.
weight1	First weight value.
weight2	Second weight value.

Discussion

The function ippsWeightedSum calculates the weighted sum as follows:

$$pDst[i] = \frac{weight1 \cdot pSrc1[i] + weight2 \cdot pSrc2[i]}{weight1 + weight2}, \quad i = 0, \ldots len - 1.$$

The function ippsWeightedSumHalf calculates the weighted sum as given by:

$$pDst[i] = \frac{pSrc1[i] + weight2 \cdot pSrc2[i]}{weight1 + weight2}, \quad i = 0, \ldots len - 1$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc1*, *pSrc2*, or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZero	Indicates a warning for zero-valued divisor vector element. The execution is not aborted. The value of the destination vector element for the floating-point operations is set as follows:

NaN	For zero-valued dividend vector element;
+Inf	For positive dividend vector element;
-Inf	For negative dividend vector element.

Vector Quantization

This section describes some functions for vector quantization and codebook operations. These functions are commonly used in acoustic and language model compressions.

FormVector

Constructs an output vector of multiple streams from codebook entries.

```
IppStatus ippsFormVector_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc, const
   Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
   Ipp16s* pDst);
IppStatus ippsFormVector_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc, const
   Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
   Ipp16s* pDst);
IppStatus ippsFormVector_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc, const
   Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
   Ipp32f * pDst);
IppStatus ippsFormVector_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc, const
   Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
   Ipp32f * pDst);

IppStatus ippsFormVector_2i_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc,
   const Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_2i_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc,
   const Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_2i_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc,
   const Ipp32s* pHeights, Ipp32f* pDst, int len);
IppStatus ippsFormVector_2i_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc,
   const Ipp32s* pHeights, Ipp32f* pDst, int len);

IppStatus ippsFormVector_4i_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc,
   const Ipp32s* pHeights, Ipp16s* pDst, int len);
```

```
IppStatus ippsFormVector_4i_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc,
   const Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_4i_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc,
   const Ipp32s* pHeights, Ipp32f* pDst, int len);
IppStatus ippsFormVector_4i_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc,
   const Ipp32s* pHeights, Ipp32f* pDst, int len);
```

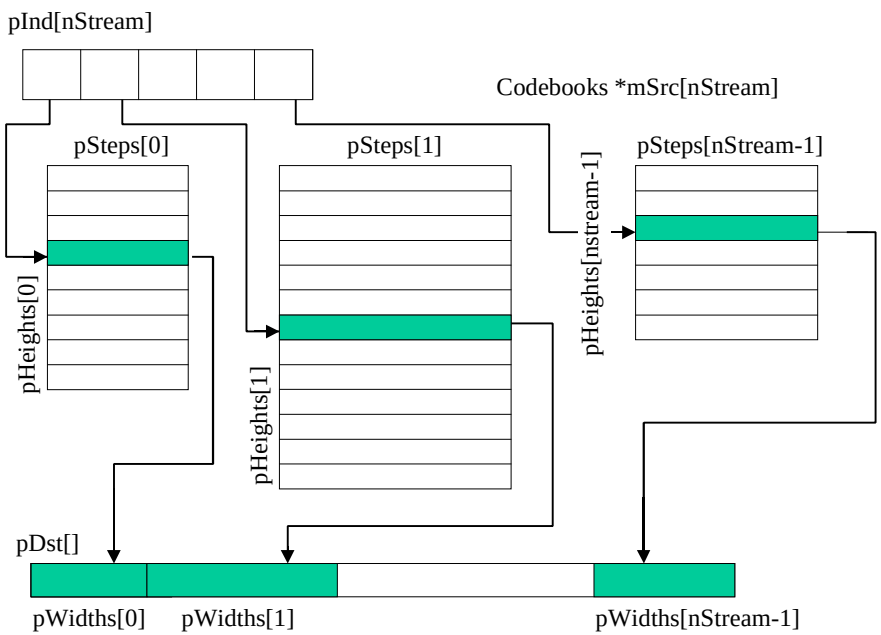
Arguments

pInd	Pointer to the indexing vector [*nStream*].
mSrc	Pointer to the array of pointers to the codebooks [*nStream*].
pHeights	Pointer to the codebook lengths [*nStream*].
pWidths	Pointer to the stream lengths [*nStream*].
pSteps	Pointer to the codevector lengths [n*Stream*].
nStream	Number of codebooks.
pDst	Pointer to the output vector.
len	Length of the output vector.

Discussion

The function ippsFormVector constructs an output vector of multiple streams. Each stream, of size *pWeights*[], is a codebook entry indexed by *pInd*[]. The codebooks are referenced by *mSrc*[], and have *pHeights*[] number of codevectors, each of which is of size *pSteps*[]. The following figure illustrates the layout:

Figure 8-3 Stream Layout for the ippsFormVector Function



The output vector is obtained as follows:

$$pDst\left[i + \sum_{j=0}^{k-1} pWidths[j] \right] = mSrc[k][pInd[k] \cdot pSteps[k] + i] \ ,$$

k = 0,...*nStream* -1; *i* = 0,...*pWidths*[*k*] -1 .

The function ippsFormVector_2i simplifies the extraction process by posting the constraints of *pWidths*[] = *pSteps*[] = 2 and *nStream* = *len*/2 .

Similarly, the function ippsFormVector_4i implies the constraints of *pWidths*[] = *pSteps*[] = 4 and *nStream* = *len*/4.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is null.
ippStsSizeErr	Indicates an error when *len*, *nStream*, *pInd*[*k*], *pWidths*[*k*], or *pSteps*[*k*] is less than or equal to 0; or when *pHeights*[*k*] is less than or equal to *pInd*[*k*].

CdbkInitAlloc

Initializes the codebook structure.

```
IppStatus ippsCdbkInitAlloc_L2_16s(IppsCdbkState_16s** pCdbk, const Ipp16s*
   pSrc, int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);
IppStatus ippsCdbkInitAlloc_L2_32f(IppsCdbkState_32f** pCdbk, const Ipp32f*
   pSrc, int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);
```

Arguments

pCdbk	Pointer to the codebook structure to be created.
pSrc	Pointer to the source vector [*height***step*].
width	Length of the input vectors.
step	Row step in the source vector *pSrc*.
height	Number of rows in the source vector *pSrc*.
cdbkSize	Size of the codebook.

hint	One of the following values:	
	IPP_CDBK_FULL	The source data are entries of a codebook, *height* should be greater or equal to *cdbkSize*. The nearest codebook entry is located through a full search.
	IPP_CDBK_KMEANS_LONG	LBG algorithm with splitting of the most extensional cluster is used for the codebook building. The nearest codebook entry is located through a logarithmical search.
	IPP_CDBK_KMEANS_NUM	LBG algorithm with splitting of the most numerous clusters is used for the codebook building. The nearest codebook entry is located through a logarithmical search.

Discussion

The function ippsCdbkInitAlloc initializes the structure that contains the codebook and additional information to be used for fast search. The structure is used during vector quantization by ippsVQ or ippsSplitVQ functions. The Euclidean distance is used to measure the similarity between two vectors.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pCdbk* or *pSrc* pointer is null.
ippStsSizeErr	Indicates an error when *width*, *step*, or *cdbkSize* is less than or equal to 0; or *cdbkSize* is greater than *height*; or *width* is greater than *step*; or *cdbkSize* is greater than 16383; or *hint* is equal to IPP_CDBK_FULL and *cdbkSize* is not equal to *height*.
ippStsCdbkFlagErr	Indicates an error when the *hint* value is incorrect.
ippStsMemAllocErr	Indicates an error when no memory was allocated.

CdbkFree

Destroys the codebook structure.

```
IppStatus ippsCdbkFree_16s(IppsCdbkState_16s* pCdbk);
IppStatus ippsCdbkFree_32f(IppsCdbkState_32f* pCdbk);
```

Arguments

pCdbk	Pointer to the codebook structure.

Discussion

The function `ippsCdbkFree` destroys the codebook structure and frees all memory associated with it.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pCdbk* pointer is null.

GetCdbkSize

Retrieves the number of codevectors in the codebook.

```
IppStatus ippsGetCdbkSize_16s(const IppsCdbkState_16s* pCdbk, int* pNum);
IppStatus ippsGetCdbkSize_32f(const IppsCdbkState_32f* pCdbk, int* pNum);
```

Arguments

pCdbk	Pointer to the codebook structure.
pNum	Pointer to the result number of codevectors.

Discussion

The function ippsGetCdbkSize retrieves the number of codevectors in the codebook *pCdbk*. This number could be less than *cdbkSize* if the number of different vectors in *pSrc* is less than *cdbkSize*. The codebook structure *pCdbk* is initialized by the ippsCdbkAlloc function.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pCdbk* or *pNum* pointer is null.

GetCodebook

Retrieves the codevectors from the codebook.

```
IppStatus ippsGetCodebook_16s(const IppsCdbkState_16s* pCdbk, Ipp16s* pDst,
   int step);
IppStatus ippsGetCodebook_32f(const IppsCdbkState_32f* pCdbk, Ipp32f* pDst,
   int step);
```

Arguments

pCdbk	Pointer to the codebook structure.
pDst	Pointer to the destination vector for codevectors [*pNum*[0]**step*].
step	Row step in the destination vector *pDst*.

Discussion

The function ippsGetCodebook retrieves the codevectors from the codebook structure *pCdbk* and stores them in the *pDst* vector with row step *step*.

The codebook structure *pCdbk* is initialized by the function ippsCdbkAlloc. The number of clusters can be obtained by the function ippsGetCdbkSize.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pCdbk* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *step* or equal to 0 or *step* is less than *width*.

VQ

Quantizes the input vectors given a codebook.

```
IppStatus ippsVQ_16s(const Ipp16s* pSrc, int step, Ipp32s* pIndx, int height,
   IppsCdbkState_16s* pCdbk);
IppStatus ippsVQ_32f(const Ipp32f* pSrc, int step, Ipp32s* pIndx, int height,
   IppsCdbkState_32f* pCdbk);
IppStatus ippsVQDist_16s32s_Sfs(const Ipp16s* pSrc, int step, Ipp32s* pIndx,
   Ipp32s* pDist, int height, IppsCdbkState_16s* pCdbk, int scaleFactor);
IppStatus ippsVQDist_32f(const Ipp32f* pSrc, int step, Ipp32s* pIndx, Ipp32f*
   pDist, int height, IppsCdbkState_32f* pCdbk);
```

Arguments

pCdbk	Pointer to the codebook structure.
pSrc	Pointer to the source vector [*height*step*].
step	Row step in the source vector *pSrc*.
height	Number of rows in the source vector *pSrc*.
pIndx	Pointer to the resulting index vector of the closest codevectors [*height*].
pDist	Pointer to the resulting quantization distances from the source vector [*height*].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsVQ performs Vector Quantization (VQ) on the input vectors. The resulting indexes of the closest codevectors are stored in the vector *pIndx*. The function ippsVQDist also stores the distances (scaled with *scaleFactor* for the integer versions) to the output distance vector *pDist*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pCdbk*, *pSrc*, *pIndx*, or *pDist* pointer is null.
ippStsSizeErr	Indicates an error when *step* or *height* is less than or equal to 0.

SplitVQ

Quantizes a multiple-stream vector given the codebooks.

```
IppStatus ippsSplitVQ_16s16s(const Ipp16s* pSrc, int srcStep, Ipp16s* pDst,
   int dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
IppStatus ippsSplitVQ_16s8u(const Ipp16s* pSrc, int srcStep, Ipp8u* pDst, int
   dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
IppStatus ippsSplitVQ_16s1u(const Ipp16s* pSrc, int srcStep, Ipp8u* pDst, int
   dstBitStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
IppStatus ippsSplitVQ_32f16s(const Ipp32f* pSrc, int srcStep, Ipp16s* pDst,
   int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
IppStatus ippsSplitVQ_32f8u(const Ipp32f* pSrc, int srcStep, Ipp8u* pDst, int
   dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
IppStatus ippsSplitVQ _32f1u(const Ipp32f* pSrc, int srcStep, Ipp8u* pDst, int
   dstBitStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
```

Arguments

pCdbks	Pointer to the codebook structures [*nStream*].

pSrc	Pointer to the source vector [*height*srcStep*].
srcStep	Row step in the source vector *pSrc*.
pDst	Pointer to the destination indexing vector [*height*dstStep*].
dstStep	Row step in the destination vector *pDst*.
height	Number of rows in the source and destination vectors.
dstBitStep	Row step in the destination vector (in bits).
nStream	Number of streams in the source vectors.

Discussion

The functions ippsSplitVQ quantizes the multiple-stream vectors *pSrc* against given codebooks *pDdbks*. The length of each stream is assumed to be equal to that of the corresponding codebook vectors. The outputs *pDst* are indexes to the codebook entries.

For functions with the 1u suffix, the output indexes are packed in bits. Each stream takes the least number of bits sufficient to represent its codebook indexes.

See Also

ippsFormVector, ippsFormVectorVQ

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pCdbk*, *pCdbk*[*k*], *pSrc*, or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *srcStep*, *dstStep*, *height*, or *nStream* is less than or equal to 0; or the sum of the stream length is greater than *srcStep;* or *nStream* is greater than *dstStep* for functions with the 16s or 8u suffix; or the number of bits sufficient to represent the indexes is greater than *dstStep* for functions with the 1u suffix; or the codebook size is greater than 256 for functions with the 8u suffix.

FormVectorVQ

Constructs multiple-stream vectors from codebooks, given indexes.

```
IppStatus ippsFormVectorVQ_16s16s(const Ipp16s* pSrc, int srcStep, Ipp16s*
   pDst, int dstStep, int height, const IppsCdbkState_16s** pCdbks, int
   nStream);
IppStatus ippsFormVectorVQ_8u16s(const Ipp8u* pSrc, int srcStep, Ipp16s* pDst,
   int dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
IppStatus ippsFormVectorVQ_1u16s(const Ipp8u* pSrc, int srcBitStep, Ipp16s*
   pDst, int dstStep, int height, const IppsCdbkState_16s** pCdbks, int
   nStream);
IppStatus ippsFormVectorVQ_16s32f(const Ipp16s* pSrc, int srcStep, Ipp32f*
   pDst, int dstStep, int height, const IppsCdbkState_32f** pCdbks, int
   nStream);
IppStatus ippsFormVectorVQ_8u32f(const Ipp8u* pSrc, int srcStep, Ipp32f* pDst,
   int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
IppStatus ippsFormVectorVQ_1u32f(const Ipp8u* pSrc, int srcBitStep, Ipp32f*
   pDst, int dstStep, int height, const IppsCdbkState_32f** pCdbks, int
   nStream);
```

Arguments

pCdbks	Pointer to the codebook structures [*nStream*].
pSrc	Pointer to the indexing vectors [*height***srcStep*].
pDst	Pointer to the constructed vectors [*height***dstStep*].
srcStep	Row step in the indexing vectors *pSrc*.
srcBitStep	Row step in the indexing vectors *pSrc* (in bits).
dstStep	Row step in the constructed vectors *pDst*.
height	Number of rows in the vectors *pSrc*.
nStream	Number of streams.

Discussion

The function ippsFormVectorVQ constructs multiple-stream vectors *pDst* from the codebooks *pCdbks* given indexes *pSrc*. The length of each stream is assumed to be equal to that of the corresponding codebook vectors.

For functions with the 1u suffix, each stream index is assumed to be in a packed format. Each stream takes the number of bits sufficient to represent its codebook indexes.

See Also

ippsFormVector, ippsSplitVQ

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pCdbk*, *pCdbk*[*k*], *pSrc*, or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *srcStep*, *dstStep*, *height*, or *nStream* is less than or equal to 0; or the codevector length sum is greater than *dstStep*; or *nStream* is greater than *srcStep* for functions with the 16s or 8u suffix; or the number of bits sufficient to represent the indexes is greater than *srcStep* for functions with the 1u suffix.

Compatibility with IPP version 1.1

Some function names of the speech recognition primitives in IPP version 1.1 are deprecated for a uniform naming convention of IPP version 2. The following table lists the deprecated functions in the left column and the renamed functions in the right column. For compatibility, the use of those deprecated functions is supported (but not recommended) in IPP version 2.

Table 8-1 Renamed Functions List

Deprecated Functions	New Function Names
ippsAddNRow	ippsAddNRows
ippsSumCol	ippsSumColumn
ippsCopyCol	ippsCopyColumn
ippsMeanCol	ippsMeanColumn
ippsVarCol	ippsVarColumn
ippsMeanVarCol	ippsMeanVarColumn
ippsNormalizeCol	ippsNormalizeColumn
ippsAddMulCol	ippsAddMulColumn
ippsQRTransCol	ippsQRTransColumn
ippsDotProdCol	ippsDotProdColumn
ippsMulCol	ippsMulColumn
ippsSumColAbs	ippsSumColumnAbs
ippsSumColSqr	ippsSumColumnSqr
ippsDeltaW1	ippsDelta_Win1
ippsDeltaW2	ippsDelta_Win2
ippsDeltaDeltaW1	ippsDeltaDelta_Win1
ippsDeltaDeltaW2	ippsDeltaDelta_Win2
ippsRecSqrt_Th	ippsRecSqrt
ippsThreshold_LM	ippsScaleLM
ippsMahDist1	ippsMahDist
ippsMahDist2	ippsMahDist_MultiMix

continued

Table 8-1 Renamed Functions List

Deprecated Functions	New Function Names
ippsLogGauss1	ippsLogGauss
ippsLogGauss2	ippsLogGauss_MultiMix
ippsLogGaussMax1	ippsLogGaussMax
ippsLogGaussAdd1	ippsLogGaussAdd
ippsLogGaussAdd2	ippsLogGaussAdd_MultiMix

Audio Coding Functions 9

The subset of IPP for audio coding includes general purpose functions applicable in several codecs and a number of specific functions for MPEG-4 audio encoder and decoder. These functions implement pipeline blocks with large computational complexity.

The current set of functions is sufficient to implement a portable optimized MPEG-4 AAC Main profile decoder.

Interleaved to Multi-row Format Conversion Functions

This section describes functions that enable you to perform transformations between interleaved and non-interleaved forms of multichannel signal. A signal in the interleaved form is a vector with interleaved samples for different channels. In the non-interleaved form, samples for each channel are stored in a separate vector.

NOTE. *Though you may use both aligned and unaligned memory for arrays, you should expect slower performance when the memory is not aligned.*

Interleave

Converts signal from non-interleaved to interleaved format.

```
IppStatus ippsInterleave_16s(const Ipp16s** pSrc, int ch_num, int len,
    Ipp16s* pDst);
IppStatus ippsInterleave_32f(const Ipp32f** pSrc, int ch_num, int len,
    Ipp32f* pDst);
```

Arguments

pSrc	Array of pointers to the vectors [*len*] containing samples for particular channels.
ch_num	Number of channels.
len	Number of samples in each channel.
pDst	Pointer to the destination vector [*ch_num* * *len*] in interleaved format.

Discussion

The function ippsInterleave transforms the signal from the non-interleaved to interleaved format according to the formula

pDst[*i*] = *pSrc*[*i* div *ch_num*][*i* mod *ch_num*], $0 \le i < len * ch\_num$, where div is the integer part of the quotient and mod is the remainder.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *length* or *ch_num* is less than or equal to 0.
ippStsMisalignedBuf	Indicates misaligned data. It is recommended to supply aligned data in order to increase performance.

Deinterleave

Converts signal from interleaved to non-interleaved format.

```
IppStatus ippsDeinterleave_16s(const Ipp16s* pSrc, int ch_num, int len,
    Ipp16s** pDst);
IppStatus ippsDeinterleave_32f(const Ipp32f* pSrc, int ch_num, int len,
    Ipp32f** pDst);
```

Arguments

pSrc	Pointer to vector [*ch_num* * *len*] of interleaved samples.
ch_num	Number of channels.
len	Number of samples in each channel.
pDst	Array of pointers to the vectors [*len*] to be filled with samples of particular channels.

Discussion

The function ippsDeinterleave transforms the input signal from interleaved to non-interleaved format according to the formula

$pDst[i][j] = pSrc[i + j * ch\_num], 0 \le i < ch\_num, 0 \le j < len.$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* or *ch_num* is less than or equal to 0.
ippStsMisalignedBuf	Indicates misaligned data. It is recommended to supply aligned data in order to increase performance.

Spectral Data Prequantization Functions

MPEG-1, 2 Layer III and MPEG-4 AAC audio encoders raise the spectral data to the power of 3/4 before quantization to provide a more consistent signal-to-noise ratio over the range of quantized values. The re-quantizers in the decoders linearize the values by raising their output to the power of 4/3.

Pow34

Raises a vector to the power of 3/4.

```
IppStatus ippsPow34_32f16s(const Ipp32f* pSrc, Ipp16s* pDst, int len);
```

Arguments

pSrc	Pointer to the input data vector [*len*].
pDst	Pointer to the output data vector [*len*].
len	Number of elements in the input and output vectors.

Discussion

The function ippsPow34 performs the calculation for each element of *pSrc* by the formula

$$pDst[i] = |pSrc[i]|^{\frac{3}{4}}, \quad 0 \le i < len,$$

and stores the result in *pDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.

Pow43

Raises a vector to the power of 4/3.

```
IppStatus ippsPow43_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the input data vector [*len*].
pDst	Pointer to the output data vector [*len*].
len	Number of elements in the input and output vectors.

Discussion

The function ippsPow43 performs the calculation for each element of *pSrc* by the formula

$$pDst[i] = |pSrc[i]|^{\frac{4}{3}}, \quad 0 \le i < len,$$

and stores the result in *pDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.

Scale Factors Calculation Functions

In MPEG-2, 4 GA AAC decoder scale factors extracted from the bit stream require an additional restoring procedure. The function `ippsCalcSF` restores actual scale factors from values transmitted in the bit stream.

CalcSF

Restores actual scale factors from the bit stream values.

```
IppStatus ippsCalcSF_16s32f(const Ipp16s* pSrc, int offset, Ipp32s* pDst,
    int len);
```

Arguments

pSrc	Pointer to the input data array.
offset	Scale factors offset.
pDst	Pointer to the output data array.
len	Number of elements in the vector.

Discussion

The function `ippsCalcSF` restores actual scale factors from the values *pSrc* transmitted in the bit stream, using the common scale factor offset. Computation is performed according to the following formula

$$pDst[i] = 2^{\frac{1}{4}(pSrc[i]-offset)}, \quad 0 \le i < len.$$

Restored scale factors are written into *pDst*.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.

Scale Factor Application Functions

This procedure is necessary for MPEG-2, 4 GA AAC decoder to restore the original spectral values from a set of the inversely quantized spectral coefficients. The whole spectrum is divided into a set of scale factor bands. Since the widths differ from band to band, the band positions are defined by the offset vector.

ApplySF_I

Applies scale factors to spectral bands in accordance with spectral bands boundaries.

```
IppStatus ippsApplySF_32f_I(Ipp32f* pSrcDst, const Ipp32f* pSF,
    const int *pBandOffset, int bands_number);
```

Arguments

pSrcDst	Pointer to the input and output data array.
pSF	Pointer to the data array containing scale factors.
pBandsOffset	Pointer to the vector of band offsets.
bands_number	Number of bands to which scale factors are applied.

Discussion

The function `ippsApplySF_I` computes scaled values from the input vector, the vector of scale factors, and the vector of band offsets. Operations are performed in-place.

This function applies the set of scale factors *pSF* (that has *bands_number* elements) to bands constructed from the input vector *pSrc* . Band boundaries are defined by the vector of the band offsets *pBandOffset*. All values in each band are multiplied by the corresponding scale factor.

NOTE. *The function operates on the assumption that the end of the last band coincides with the end of the spectral data vector, that is, the size of* `pSrcDst` *vector is contained in the* `pBandsOffset[bands_number]` *element.*

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrc`* or *`pDst`* pointer is `NULL`.

Modified Discrete Cosine Transform Functions

This section describes IPP functions that compute the modified discrete cosine transform (MDCT) of a signal. MDCT is a lapped orthogonal transform widely used in different audio codecs, such as MPEG-1, MPEG-2, AC-3, and AAC.

MDCTFwdInitAlloc, MDCTInvInitAlloc

Initializes modified discrete cosine transform specification structure.

```
IppStatus ippsMDCTFwdInitAlloc_32f(IppsMDCTFwdSpec_32f** pMDCTSpec,
    int length);
IppStatus ippsMDCTInvInitAlloc_32f(IppsMDCTInvSpec_32f** pMDCTSpec,
    int length);
```

Arguments

`pMDCTSpec`	Pointer to MDCT specification structure to be created.

length	Number of samples in MDCT. Since this set of functions was designed specially for audio coding, only the following values of length are supported: 12, 36, and 2^k, where $k \geq 5$. These values are the only values that appear in audio coding.

Discussion

The functions ippsMDCTFwdInitAlloc and ippsMDCTInvInitAlloc create and initialize MDCT specification structure *pMDCTSpec* with the specified transform length *length*.

ippsMDCTFwdInitAlloc. The function ippsMDCTFwdInitAlloc initializes the forward MDCT specification structure.

ippsMDCTInvInitAlloc. The function ippsMDCTInvInitAlloc initializes the inverse MDCT specification structure.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMDCTSpec* pointer is null.
ippStsSizeErr	Indicates an error when *length* does not belong to the above set of admissible values.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

MDCTFwdFree, MDCTInvFree

Closes modified discrete cosine transform specification structure.

```
IppStatus ippsMDCTFwdFree_32f(IppsMDCTFwdSpec_32f* pMDCTSpec);
IppStatus ippsMDCTInvFree_32f(IppsMDCTInvSpec_32f* pMDCTSpec);
```

Arguments

pMDCTSpec	Pointer to the MDCT specification structure to be closed.

Discussion

The functions ippsMDCTFwdFree and ippsMDCTInvFree close the MDCT structure *pMDCTSpec* by freeing all memory associated with the specification created by ippsMDCTFwdInitAlloc or ippsMDCTInvInitAlloc functions.

Call either ippsMDCTFwdFree or ippsMDCTInvFree after the transform is completed.

ippsMDCTFwdFree. The function ippsMDCTFwdFree closes the forward MDCT specification structure.

ippsMDCTInvFree. The function ippsMDCTInvFree closes the inverse MDCT specification structure.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMDCTSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pMDCTSpec* is incorrect.

MDCTFwdGetBufSize, MDCTInvGetBufSize

Gets the size of MDCT work buffer.

```
IppStatus ippsMDCTFwdGetBufSize_32f(const IppsMDCTFwdSpec_32f* pMDCTSpec, int*
    pSize);
IppStatus ippsMDCTInvGetBufSize_32f(const IppsMDCTInvSpec_32f* pMDCTSpec, int*
    pSize);
```

Arguments

pMDCTSpec	Pointer to the MDCT specification structure.
pSize	Pointer to the MDCT work buffer size value (in bytes).

Discussion

The functions `ippsMDCTFwdGetBufSize` and `ippsMDCTInvGetBufSize` get the work buffer size of the MDCT described by the specification structure *pMDCTSpec* and store the result in *pSize*.

ippsMDCTFwdGetBufSize. The function `ippsMDCTFwdGetBufSize` gets the size of the work buffer for the forward MDCT.

ippsMDCTInvGetBufSize. The function `ippsMDCTInvGetBufSize` gets the size of the work buffer for the inverse MDCT.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMDCTSpec* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification structure *pMDCTSpec* is invalid.

MDCTFwd, MDCTInv

Computes forward or inverse modified discrete cosine transform (MDCT) of a signal.

```
IppStatus ippsMDCTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst,const
    IppsMDCTFwdSpec_32f* pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst,const
    IppsMDCTInvSpec_32f* pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTFwd_32f_I(Ipp32f* pSrcDst, const IppsMDCTFwdSpec_32f*
    pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTInv_32f_I(Ipp32f* pSrcDst, const IppsMDCTInvSpec_32f*
    pMDCTSpec, Ipp8u* pBuffer);
```

Arguments

pSrc	Pointer to the input data array.
pDst	Pointer to the output data array.
pSrcDst	Pointer to the input and output data array for the in-place operations.
pMDCTSpec	Pointer to the MDCT specification structure.
pBuffer	Pointer to the MDCT work buffer.

Discussion

The functions ippsMDCTFwd and ippsMDCTInv compute the forward and inverse modified discrete cosine transform (MDCT), respectively.
In the following definition of MDCT, N denotes the length and $n_0 = (N/2+1)/2$.

For the forward MDCT, $x(n)$ is *pSrc*[*n*] and $y(k)$ is *pDst*[*k*], whereas for the inverse MDCT $x(n)$ is *pDst*[*n*] and $y(k)$ is *pSrc*[*k*].

The forward MDCT is defined by the formula:

$$y(k) = 2 \cdot \sum_{n=0}^{N-1} x(n) \cos\left(\frac{\pi}{N}(n+n_0)(2k+1)\right), \text{ for } 0 \le k < \frac{N}{2}.$$

The inverse MDCT is defined as

$$x(n) = \frac{2}{N} \cdot \sum_{k=0}^{\frac{N}{2}-1} y(k) \cos\left(\frac{\pi}{N}(n+n_0)(2k+1)\right), \text{ for } 0 \le n < N.$$

The *pBuffer* argument provides the MDCT functions with the necessary work memory and helps to avoid memory allocation within the functions. The buffer may also increase the performance if the MDCT functions use the result of the previous operation stored in cache as an input array.

Return Value

ippStsNoErr	Indicates no error.

`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, or *pMDCTSpec* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification structure *pMDCTSpec* is invalid.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.
`ippStsMisalignedBuf`	Indicates misaligned data. It is recommended to supply aligned data in order to increase performance.

Block Filtering Functions

The IPP functions described in this section implement the finite impulse response (FIR) block filter. You can use this group of functions to design transform domain adaptive filters. These filters preprocess the signal by decomposing the input vector into orthogonal components, which are subsequently used as inputs to a parallel bank of adaptive subfilters. You may use this approach for implementing frequency domain linear predictors in audio codecs, for example, CELP and AAC.

The filtering function receives a number of vectors (signals). Every call of a filtering function produces one filtered sample for each input signal. The library functions do not perform any particular adaptation method but you can specify the filter taps at each call of a filtering function.

To use the FIR block filter functions, follow these general steps:

1. Call `ippsFIRBlockInitAlloc` to initialize the state structure of a block filter.
2. Call `ippsFIRBlockOne` to filter a vector of samples through a block filter.
3. Call `ippsFIRBlockFree` to free dynamic memory associated with the FIR block filter.

FIRBlockInitAlloc

Initializes FIR block filter state.

```
IppStatus ippsFIRBlockInitAlloc_32f(IppsFIRBlockState_32f** pState, int order,
    int len);
```

Arguments

pState	Pointer to the FIR block filter state structure to be created.
order	Number of elements in the array containing the tap values.
len	Number of input signals.

Discussion

The function `ippsFIRBlockInitAlloc` creates and initializes a FIR block filter state.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are `NULL`.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.
`ippStsFIRLenErr`	Indicates an error when *order* or *len* is less than or equal to 0.

FIRBlockFree

Closes FIR block filter state.

```
IppStatus ippsFIRBlockFree_32f(IppsFIRBlockState_32f* pState);
```

Arguments

pState	Pointer to the FIR block filter state structure to be closed.

Discussion

The function ippsFIRBlockFree closes the FIR block filter state by freeing all memory associated with the filter state created by the function ippsFIRBlockInitAlloc.

Call ippsFIRBlockFree after filtering is completed.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state structure is invalid.

FIRBlockOne

Filters vector of samples through FIR block filter.

```
IppStatus ippsFIRBlockOne_32f(Ipp32f* pSrc, Ipp32f* pDst,
    IppsFIRBlockState_32f* pState, Ipp32f *pTaps);
```

Arguments

pSrc	Pointer to the input vector of samples to be filtered.
pDst	Pointer to the vector of filtered output samples.
pState	Pointer to the FIR filter state structure.
pTaps	Pointer to the vector of filter taps.

Discussion

The function ippsFIRBlockOne filters a vector of samples *pSrc* of the length *len* through a filter and stores the result in *pDst*.

The filter taps are specified in the vector *pTaps* of the length *order*. The values of *len* and *order* parameters are specified in the ippsFIRBlockInitAlloc call.

In the following definition of the FIR filter, the sample of the input vector i to be filtered with the delay k is denoted x_{n-k}^{i}, and the taps are denoted h_k. The output value y_i is defined by the following formula:

$$y_n^i = \sum_{k=0}^{order-1} h_k x_{n-k}^i, \quad 0 \le i < len.$$

Before calling the function `ippsFIRBlockOne`, initialize the filter state by calling the function `ippsFIRBlockInitAlloc`. Specify the taps values in the argument *pTaps*.

Example 9-1 Single-Rate Filtering with the ippsFIRBlockOne Function

```
IppStatus fir(void)
{

    #undef NUMITERS
    #define  NUMITERS 20

    #undef BLOCKSIZE
    #define  BLOCKSIZE 20

    int n;
    int i;
    IppStatus status;
    IppsFIRBlockState_32f *fctx;

    Ipp32f x [BLOCKSIZE],
    y = [BLOCKSIZE],
    const float taps[] = {
    0.0051f, 0.0180f, 0.0591f, 0.1245f, 0.1869f, 0.2127f, 0.1869f,
    0.1245f, 0.0591f, 0.0180f, 0.0051f, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0
    };

    ippsFIRBlockInitAlloc_32f( &fctx, 11, BLOCKSIZE );
    for (n =0;n<NUMITERS;++n)
    {
        for (i = 0; i< BLOCKSIZE; i++) x[i]=(float)sin(IPP_2PI *
            n *0.2 + i);
        status = ippsFIRBlockOne_32f( x, y, fctx, taps );
        for (i = 0; i< BLOCKSIZE; i++) printf_32f("%f ", y[i]);
    }
    ippsFIRBlockFree_32f(fctx);
    return status;
}
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state structure is invalid.
ippStsMisalignedBuf	Indicates misaligned arrays. It is recommended to supply aligned data in order to increase performance.

Frequency Domain Prediction Functions

MPEG-2, 4 AAC encoder uses prediction in frequency domain (FDP) to decrease redundancy in the audio signal and ensure more effective coding. For each spectral line, input signals are filtered through the second order adaptive FIR filter called a predictor. Then, instead of processing the signal, the difference between the original signal and the filter output (that is, the prediction error), is passed for further processing. The decoder derives the original signal from the prediction error using a symmetrical block.

You should regularly reset the predictors to their initial state to reduce accumulated calculation error. You may also need to reset the predictors in special cases discussed and described in the ISO-144963 standard. You can reset predictors for the entire spectrum, several scale factor bands, or a selected group of spectral lines.

For more information on the algorithm of filter coefficient adaptation and FDP usage see ISO-144963, clause 6.5.3.2.

To use the FDP prediction tool functions described in this section, follow these steps:

1. Call the function ippsFDPInitAlloc to allocate memory and initialize predictor state.
2. Call the function ippsFDPFwd for each frame to calculate prediction error or call the function ippsFDPInv to retrieve the original signal.
3. Call the functions ippsResetFDP, ippsResetFDP_SFB, or ippsResetFDPGroup to reset predictors in certain spectral lines at any time after creating the state.
4. Call the function ippsFDPFree to free the memory allocated by ippsFDPInitAlloc.

FDPInitAlloc

Initializes predictor state.

```
IppStatus ippsFDPInitAlloc_32f(IppsFDPState_32f **pFDPState, int len);
```

Arguments

pFDPState	Pointer to the FDP state structure to be created.
len	Number of spectral lines to be processed.

Discussion

The function `ippsFDPInitAlloc` creates and initializes the FDP state.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one or more pointers passed to function is `NULL`.
`ippStsSizeErr`	Indicates an error when the length is less than or equal to 0.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

FDPFree

Closes FDP state.

```
IppStatus ippsFDPFree_32f(IppsFDPState_32f *pFDPState);
```

Arguments

pFDPState	Pointer to the FDP state structure to be closed.

Discussion

The function ippsFDPFree closes the FDP state by freeing all memory associated with the FDP state structure created by the function ippsFIRBlockInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one or more pointers passed to the function is NULL.
ippStsContextMatchErr	Indicates an error when the state structure is invalid.

ResetFDP

Resets predictors for all spectral lines.

```
IppStatus ippsResetFDP_32f(IppsFDPState_32f *pFDPState);
```

Arguments

pFDPState	Pointer to the predictor specific state structure.

Discussion

The function ippsResetFDP resets predictors for all spectral lines.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one or more pointers passed to the function is NULL.
ippStsContextMatchErr	Indicates an error when the state structure is invalid.

ResetFDP_SFB

Resets predictor-specific information in some scale factor bands.

```
IppStatus ippsResetFDP_SFB_32f (IppsFDPState_32f* pFDPState, const int*
    pBandsOffset, int bands_number, const Ipp8u *reset_flag);
```

Arguments

pFDPState	Pointer to the predictor specific state structure.
pBandsOffset	Pointer to band offset vector.
bands_number	Number of bands.
reset_flag	Array of flags showing whether predictors for spectral lines in a certain scale factor band need to be reset.

Discussion

The function `ippsResetFDP_32f` resets predictors for all spectral lines in each scale factor band *i*, for which *reset_flag*[*i*] is not equal to 0.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one or more pointers passed to the function is `NULL`.
`ippStsSizeErr`	Indicates an error when *bands_number* is less than or equal to 0.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.

ResetFDPGroup

Resets predictors for group of spectral lines.

```
IppStatus ippsResetFDPGroup_32f (IppsFDPState_32f* pFDPState, int start, int
    step);
```

Arguments

pFDPState	Pointer to the predictor specific state structure.
start	Offset of the first spectral line in the group.
step	The distance between two neighbor spectral lines in the group.

Discussion

The function `ippsResetFDPGroup` resets predictors for each *step*-th spectral line beginning from the start up to the end of spectrum.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one or more pointers passed to the function is null.
`ippStsSizeErr`	Indicates an error when *reset_group_number* or *step* is less than or equal to 0.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.

FDPFwd

Performs frequency domain prediction procedure and calculates prediction error.

```
IppStatus ippsFDPFwd_32f(IppsFDPState_32f* pFDPState, Ipp32f* pSrc,
    Ipp32f* pDst);
```

Arguments

pFDPState	Pointer to the predictor specific state structure.
pSrc	Pointer to the input data array.
pDst	Pointer to the data array to be filled with prediction errors.

Discussion

The function `ippsFDPFwd` applies frequency domain prediction procedure to the input signal *pSrc* and stores prediction errors in the *pDst* vector.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one or more pointers passed to the function is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.
`ippStsMisalignedBuf`	Indicates misaligned arrays. It is recommended to supply aligned data in order to increase performance.

FDPInv

Retrieves input signal from prediction error, using frequency domain prediction procedure.

```
IppStatus ippsFDPInv_32f(IppsFDPState_32f* pFDPState, Ipp32f* pSrcDst, const
    int* pBandsOffset, int bands_number, const Ipp8u* prediction_used);
```

Arguments

pFDPState	Pointer to the predictor specific state structure.
pSrcDst	Pointer to the input and output data array for the in-place operation.
pBandsOffset	Pointer to the band offset vector.
bands_number	Number of scale factor bands.
prediction_used	Array of flags showing whether prediction will be used in certain scale factor band.

Discussion

The function ippsFDPInv applies the procedure of frequency domain prediction to specific bands of the input spectral vector *pSrcDst*. Positions of bands are defined by the parameters *bands_number* and *pBandsOffset*.

For each scale factor band *i*, if *prediction_used*[*i*] is not equal to 0, all values of *pSrcDst* within the band are treated as a prediction error and the original signal is restored. If *prediction_used*[*i*] is equal to 0, all values of *pSrcDst* within the band are treated as a signal and will be passed without any changes.

Regardless of the *prediction_used* flag , the coefficients of each predictor are updated.

NOTE. *The function operates on the assumption that the end of the last band coincides with the end of the spectral data vector, that is, the size of* `pSrcDst` *vector is stored in the* `pBandsOffset[bands_number]` *element.*

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one or more pointers passed to the function is `NULL`.
`ippStsSizeErr`	Indicates an error when *`bands_number`* is less than or equal to 0.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.
`ippStsMisalignedBuf`	Indicates misaligned arrays. It is recommended to supply aligned data in order to increase performance.

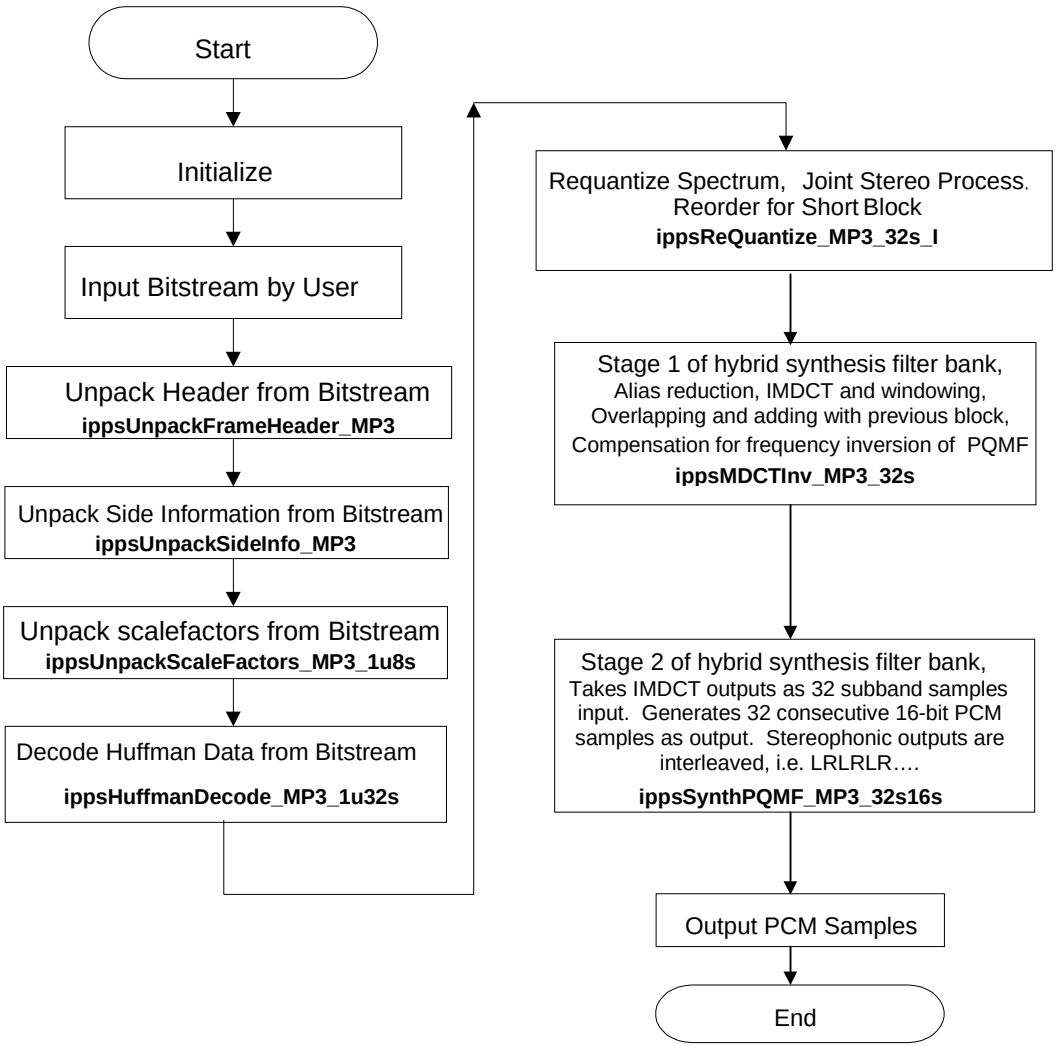
MP3 Audio Decoder

10

This chapter describes the IPP functions that can be used to construct audio decoders compliant with the layer III portions of the ISO/IEC 11172-3 MPEG-1 and ISO/IEC 13818-3 MPEG-2 audio specifications. These international standards for perceptual coding (compression) of digital audio are most often denoted by the “MP3” acronym. The MP3 algorithm delivers high quality audio with bit rates as low as one-tenth of the original, making it a popular choice for cost-sensitive and bandwidth-constrained transmission and/or storage applications. At the same time, the MP3 decoder is characterized by a manageable computational complexity. As a result, MP3 has become the de facto standard audio compression technology for emergent portable and hand-held digital media, as well as for distribution of high-fidelity compressed audio over networks such as the Internet.

This chapter provides a programmer’s reference guide to the Intel IPP MP3 Application Programming Interface (API). As shown in Figure 10-1, the MP3 API consists of seven functions, two data structures, and several pre-defined constants and macros.

Figure 10-1 MP3 Decoder API



Macros and Constants

The MP3 macro and constant definitions are listed in Table 10-1.

Table 10-1 MP3 Macro and Constant Definitions

Global Macro Name	Definition	Notes
IPP_MP3_GRANULE_LEN	576	The number of samples in one granule.
IPP_MP3_V_BUF_LEN	512	V data buffers length (32-bit words).
IPP_MP3_SF_BUF_LEN	40	Scalefactor buffer length (8-bit words).
IPP_MP3_SFB_TABLE_LONG_LEN	138	Scalefactor band table for long block length (16-bit words).
IPP_MP3_SFB_TABLE_SHORT_LEN	84	Scalefactor band table for short block length (16-bit words).

Data Structures

The decoder API includes two data structures. The structure `IppMP3FrameHeader` contains the complete set of header information associated with one frame. The structure `IppMP3SideInfo` contains the complete set of side information associated with one granule of one channel.

Frame Header

```
typedef struct {
    int id;            /* ID 1: MPEG-1, 0: MPEG-2 */
    int layer;         /* layer index 0x3: Layer I
                       //             0x2: Layer II
                       //             0x1: Layer III */
    int protectionBit; /* CRC flag 0: CRC on, 1: CRC off */
    int bitRate;       /* bit rate index */
    int samplingFreq;  /* sampling frequency index */
    int paddingBit;    /* padding flag 0: no padding, 1 padding */
    int privateBit;    /* private_bit, not used */
    int mode;          /* mono/stereo selection */
```

```
    int modeExt;       /* extension to mode */
    int copyright;     /* copyright or not, 0: no, 1: yes */
    int originalCopy;  /* original or copied, 0: copy, 1: original*/
    int emphasis;      /* flag indicating the type of de-emphasis */
    int CRCWord;       /* CRC-check word */

} IppMP3FrameHeader;
```

Side Information

```
typedef struct {
    int  part23Len;    /* number of main_data bits */
    int  bigVals;      /* half the number of Huffman code words whose
                          maximum amplitudes may be greater than 1 */
    int  globGain;     /* quantizes step size information */
    int  sfCompress;   /* number of bits used for scale factors */
    int  winSwitch;    /* window switch flag */
    int  blockType;    /* block type flag */
    int  mixedBlock;   /* flag 0: non mixed block, 1: mixed block */
    int pTableSelect[3];  /* Huffman table index for the 3 rectangle in
                             <big_values> field */
    int  pSubBlkGain[3]; /* gain offset from the global gain for one
                            subblock */
    int  reg0Cnt;      /* the number of scale factor bands in the
                          first region of <big_values> less one */
    int  reg1Cnt;      /* the number of scale factor bands in the
                          second region of <big_values> less one */
    int  preFlag;      /* flag indicating high frequency boost */
    int  sfScale;      /* scalefactor scaling */
    int  cntlTabSel;   /* Huffman table index for the <count1> field
                          of quadruples */
 } IppMP3SideInfo;
```

MP3 Audio Decoder Functions

Table 10-2 lists all MP3 audio decoder functions described in more detail later in this section.

Table 10-2 MP3 Audio Decoder Functions

Function Base Name	Description
UnpackFrameHeader_MP3	Unpacks the audio frame header.
UnpackSideInfo_MP3	Unpacks the side information from the input bitstream for use during the decoding of the associated frame.
UnpackScaleFactors_MP3	Unpacks scalefactors.
HuffmanDecode_MP3	Decodes Huffman data.
ReQuantize_MP3	Requantizes the decoded Huffman symbols.
MDCTInv_MP3	Performs stage 1 of hybrid synthesis filter bank.
SynthPQMF_MP3	Performs stage 2 of hybrid synthesis filter bank.

UnpackFrameHeader_MP3

Unpacks the audio frame header.

```
IppStatus ippsUnpackFrameHeader_MP3(Ipp8u** ppBitStream,
     IppMP3FrameHeader* pFrameHeader);
```

Arguments

ppBitStream	Pointer to pointer to the first byte of the MP3 frame header (**ppBitstream* will be updated in the function).
pFrameHeader	Pointer to the MP3 frame header structure.

Discussion

The function ippsUnpackFrameHeader_MP3 unpacks the audio frame header. If cyclic redundancy check (CRC) is enabled, this function also unpacks the CRC word. Before calling ippsUnpackFrameHeader_MP3, the user must locate the bit stream synchronization word and ensure that *ppBitStream points to the first byte of the 32-bit frame header. If CRC is enabled, it is assumed that the 16-bit CRC word is adjacent to the 32-bit frame header, as defined in the MP3 standard. Before returning to the caller, the function updates the pointer *ppBitStream such that it references the next byte after the frame header or the CRC word. The first byte of the 16-bit CRC word is stored in pFrameHeader->CRCWord(15:8), and the second byte is stored in pFrameHeader->CRCWord(7:0). The function does not detect corrupted frame headers.

Return Value

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error condition if at least one of the specified pointers is NULL.

UnpackSideInfo_MP3

Unpacks the side information from the input bitstream for use during the decoding of the associated frame.

```
IppStatus ippsUnpackSideInfo_MP3(Ipp8u** ppBitStream,
     IppMP3SideInfo* pDstSideInfo, int* pDstMainDataBegin,
     int* pDstPrivateBits, int* pDstScfsi,
     IppMP3FrameHeader* pFrameHeader);
```

Arguments

ppBitStream	Pointer to pointer to the first byte of the side information associated with the current frame in the bit stream buffer (*ppBitstream will be updated in the function).

pFrameHeader	Pointer to the structure that contains the unpacked MP3 frame header. The header structure provides format information about the input bitstream. Both single- and dual-channel MPEG-1 and MPEG-2 modes are supported.
pDstSideInfo	Pointer to the MP3 side information structure. The structure contains side information that applies to all granules and all channels for the current frame. One or more of the structures are placed contiguously in the buffer pointed by *pDstSideInfo* in the following order: {granule 0 (channel 0, channel 1), granule 1 (channel 0, channel 1)}.
pDstMainDataBegin	Pointer to the `main_data_begin` field.
pDstPrivateBits	Pointer to the `private bits` field.
pDstScfsi	Pointer to the scalefactor selection information associated with the current frame. The data is organized contiguously in the buffer pointed to by *pDstScfsi* in the following order: {channel 0 (scfsi_band 0, scfsi_band 1, ..., scfsi_band 3), channel 1 (scfsi_band 0, scfsi_band 1, ..., scfsi_band 3)}.

Discussion

The function `ippsUnpackSideInfo_MP3` unpacks the side information from the input bitstream. Before `ippsUnpackSideInfo_MP3` is called, the pointer **ppBitStream* must point to the first byte of the bit stream that contains the side information associated with the current frame. Before returning to the caller, the function updates the pointer **ppBitStream* such that it references the next byte after the side information.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsErr`	Indicates an error condition if one or more elements of the MP3 frame header structure are invalid, that is, one or more of the following conditions is true: *pFrameHeader*`->id` exceeds [0,1]; *pFrameHeader*`->layer!=1`;

	pFrameHeader->mode exceeds [0,3]; block_type has a zero value when window_switching_flag is set.

UnpackScaleFactors_MP3

Unpacks scalefactors.

```
IppStatus ippsUnpackScaleFactors_MP3_1u8s(Ipp8u** ppBitStream,
     int* pOffset, Ipp8s* pDstScaleFactor, IppMP3SideInfo* pSideInfo,
     int* pScfsi, IppMP3FrameHeader* pFrameHeader, int granule,
     int channel);
```

Arguments

ppBitStream	Pointer to pointer to the first bitstream buffer byte that is associated with the scalefactors for the current frame, granule, and channel (**ppBitStream* will be updated in the function).
pOffSet	Pointer to the next bit in the byte referenced by **ppBitStream*. Valid within the range of 0 to 7, where 0 corresponds to the most significant bit and 7 corresponds to the least significant bit (**pOffSet* will be updated in the function).
pSideInfo	Pointer to the MP3 side information structure associated with the current granule and channel.
pScfsi	Pointer to scalefactor selection information for the current channel.
channel	Channel index; can take on the values of either 0 or 1.
granule	Granule index; can take on the values of either 0 or 1.
pFrameHeader	Pointer to MP3 frame header structure for the current frame.
pDstScaleFactor	Pointer to the scalefactor vector for long and/or short blocks.

Discussion

The function ippsUnpackScaleFactors_MP3 unpacks short and/or long block scalefactors for one granule of one channel and places the results in the vector *pDstScaleFactor*. Before returning to the caller, the function updates **ppBitStream* and **pOffset* such that they point to the next available bit in the input bitstream.

NOTE. *MPEG-2 intensity mode: if the intensity position is equal to the maximum value of intensity position (an illegal position), the illegal position will be set to negative. Thus, in the requantization module, negative positions indicate illegal positions. Those scalefactors that are not treated as intensity positions must be set to positive before using them.*

Return Value

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error condition if one or more of the following pointers is NULL: *ppBitStream*, *pOffset*, *pDstScaleFactor*, *pSideInfo*, *pScfsi*, **ppBitStream*, or *pFrameHeader*; or the value of *pOffset* is outside the range [0,7]; or the granule or channel indices have values other than 0 or 1.
ippStsErr	Indicates an error condition if one or more of the following are true: *FrameHeader*->id exceeds [0,1]; *pSideInfo*->blockType exceeds [0,3]; *pSideInfo*->mixedBlock exceeds [0,1]; if *pFrameHeader*->id indicates that the bitstream is MPEG-1, and *pSideInfo*->sfCompress exceeds [0,15] or *pScfsi* [0..3] exceeds [0,1]; if *pFrameHeader*->id indicates that the bitstream is MPEG-2, and *pSideInfo*->sfCompress exceeds [0,511] or *pFrameHeader*->modeExt exceeds [0, 3].

HuffmanDecode_MP3

Decodes Huffman data.

```
IppStatus ippsHuffmanDecode_MP3_1u32s(Ipp8u** ppBitStream,
     int* pOffset, Ipp32s* pDstIs, int* pDstNonZeroBound,
     IppMP3SideInfo* pSideInfo, IppMP3FrameHeader* pFrameHeader,
     int hufSize);
```

Arguments

ppBitStream	Pointer to pointer to the first bit stream byte that contains the Huffman codewords associated with the current granule and channel (**ppBitStream* will be updated in the function).
pOffset	Pointer to the starting bit position in the bit stream byte pointed by **ppBitStream*; valid within the range of 0 to 7, where 0 corresponds to the most significant bit, and 7 corresponds to the least significant bit (**pOffSet* will be updated in the function).
pSideInfo	Pointer to MP3 structure that contains the side information associated with the current granule and channel.
pFrameHeader	Pointer to MP3 structure that contains the header associated with the current frame.
pDstIs	Pointer to the vector of decoded Huffman symbols used to compute the quantized values of the 576 spectral coefficients that are associated with the current granule and channel.
pDstNonZeroBound	Pointer to the spectral region above which all coefficients are set equal to zero.
hufSize	The number of Huffman code bits associated with the current granule and channel.

Discussion

The function ippsHuffmanDecode_MP3 decodes Huffman symbols for the 576 spectral coefficients associated with one granule of one channel. Before returning to the caller, the function updates *ppBitStream such that it points to the particular byte in the bit stream that contains the first new bit following the decoded block of Huffman codes.

Return Value

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error condition if at least one of the specified pointers is NULL; or if *pOffset* is outside the range [0,7].
ippStsErr	Indicates an error condition if the *hufSize* value is less then 0 or greater than *pSideInfo*->part23Len. This status code can also indicate either that one or more elements of the MP3 side information are invalid or that one or more elements of the MP3 frame header are invalid (see Table 10-3 below).

Table 10-3 ippStsErr Error List for the ippsHuffmanDecode_MP3 Function

Input Data	Invalid Value	Condition
pSideInfo->bigVals * 2	>IPP_MP3_GRANULE_LEN	None
pSideInfo->bigVals * 2	< 0	None
pSideInfo->winSwitch	Exceeds [0,1]	None
pSideInfo->blockType	Exceeds [0,3]	None
pSideInfo->blockType	==0	1 == *pSideInfo*->winSwitch
pSideInfo->cnt1TabSel	Exceeds [0,1]	None
pSideInfo->reg0Cnt	< 0	0 == *pSideInfo*->blockType
pSideInfo->reg1Cnt	< 0	0 == *pSideInfo*->blockType
pSideInfo->reg0Cnt + + *pSideInfo* ->reg1Cnt + 2	> 22	0 == *pSideInfo*->blockType
pSideInfo->pTableSelect [0]	Exceeds [0,31]	None
pSideInfo->pTableSelect [1]	Exceeds [0,31]	None

Table 10-3 `ippStsErr` Error List for the `ippsHuffmanDecode_MP3` Function

Input Data	Invalid Value	Condition
pSideInfo->pTableSelect [2]	Exceeds [0,31]	0 == *pSideInfo*->blockType
pFrameHeader->id	Exceeds [0,1]	None
pFrameHeader->layer	! = 1	None
pFrameHeader->samplingFreq	Exceeds [0,2]	None
hufSize	Exceeds [0,*pSideInfo*->part23Len]	None

ReQuantize_MP3

Requantizes the decoded Huffman symbols.

```
IppStatus ippsReQuantize_MP3_32s_I(Ipp32s* pSrcDstIsXr,
     int* pNonZeroBound, Ipp8s* pScaleFactor,
     IppMP3SideInfo* pSideInfo, IppMP3FrameHeader* pFrameHeader,
     Ipp32s* pBuffer);
```

Arguments

pSrcDstIsXr	Pointer to the vector of decoded Huffman symbols; for stereo and dual_channel modes, right channel data begins at the address `&(`*pSrcDstIsXr*`[576])`. This vector will be updated in the function.
pNonZeroBound	Pointer to the spectral bound above which all coefficients are set to zero; for stereo and dual_channel modes, the left channel bound is `pNonZeroBound [0]`, and the right channel bound is `pNonZeroBound [1]`.
pScaleFactor	Pointer to the scalefactor buffer; for stereo and dual_channel modes, the right channel scalefactors begin at `& (`*pScaleFactor* `[IPP_MP3_SF_BUF_LEN])`.
pSideInfo	Pointer to the side information for the current granule.
pFrameHeader	Pointer to the frame header for the current frame.

pBuffer	Pointer to a workspace buffer; the buffer length must be 576 samples.

Discuss

The function ippsReQuantize_MP3 requantizes the decoded Huffman symbols. Spectral samples for the synthesis filter bank are derived from the decoded symbols using the requantization equations given in the ISO standard. Stereophonic mid/side (M/S) and/or intensity decoding is applied if necessary. Requantized spectral samples are returned in the vector *pSrcDstIsXr*. The reordering operation is applied for short blocks. Users must preallocate a workspace buffer pointed to by *pBuffer* prior to calling the requantization function. For short blocks, the value pointed by *pNonZeroBound* will be recalculated according to the reordered sequence.

Return Value

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsErr	Indicates that one or more of the input error conditions listed in Table 10-4 is detected:

Table 10-4 ippStsErr Error List for the ippsReQuantize_MP3 Function

Input Data	Invalid Value	Condition
pNonZeroBound [ch]	Exceeds [0,576]	None
pFrameHeader->id	Exceeds [0,1]	None
pFrameHeader->samplingFreq	Exceeds [0,2]	None
pFrameHeader->mode	Exceeds [0,3]	None
pSideInfo [ch]. blockType	Exceeds [0,3]	None
pFrameHeader->modeExt	Exceeds [0,3]	None
pSideInfo [ch]. mixedBlock	Exceeds [0,1]	None
pSideInfo [ch] .globGain	Exceeds [0,255]	None
pSideInfo [ch]. sfScale	Exceeds [0,1]	None
p*SideInfo* [ch]. preFlag	Exceeds [0,1]	None
pSideInfo [ch]. pSubBlkGain [w]	Exceeds [0,7]	None

Table 10-4 `ippStsErr` Error List for the `ippsReQuantize_MP3` Function

Input Data	Invalid Value	Condition
pSrcDstIsXr[i]	> 8206	None
pScaleFactor [sfb]	> 7	If *pScaleFactor* [sfb] is the intensity position for MPEG-1.
pSideInfo [ch]. blockType	*pSideInfo* [0]. blockType!= *pSideInfo* [1]. blockType	If the bitstream is joint stereo mode.
pSideInfo [ch]. mixedBlock	*pSideInfo* [0]. mixedBlock!= *pSideInfo* [1]. mixedBlock	If the bitstream is joint stereo mode.

NOTE. *In the above list, the range on* `ch` *is from 0 to* `chNum`*, and the range on* `w` *is from 0 to 2, where* `chNum` *is the number of channels decoded by the* `pFrameHeader->mode`*. If* `pFrameHeader->mode == 3` *then* `chNum = 1`*, otherwise* `chNum = 2`*.*

MDCTInv_MP3

Performs the first stage of hybrid synthesis filter bank.

```
IppStatus ippsMDCTInv_MP3_32s(Ipp32s* pSrcXr, Ipp32s* pDstY,
     Ipp32s* pSrcDstOverlapAdd, int nonZeroBound,
     int* pPrevNumOfImdct, int blockType, int mixedBlock);
```

Arguments

pSrcXr	Pointer to the vector of requantized spectral samples for the current channel and granule, represented in Q5.26 format.
pDstY	Pointer to the vector of IMDCT outputs in Q7.24 format, for input to PQMF bank.

pSrcDstOverlapAdd	Pointer to the overlap-add buffer; contains the overlapped portion of the previous granule's IMDCT output, in Q7.24 format. This buffer will be updated in the function.
nonZeroBound	The bound above which all spectral coefficients are zero for the current granule and channel.
pPrevNumOfImdct	Pointer to the number of IMDCTs computed for the current channel of the previous granule, it will be updated in the function such that it will reference the number of IMDCTs for current granule.
blockType	Block type indicator.
mixedBlock	Mixed block indicator.

Discussion

The function `ippsMDCTInv_MP3` performs the first stage of the hybrid synthesis filter bank. The following operations are performed:

a) Alias reduction,

b) Inverse modified discrete cosine transform (IMDCT) according to block size specifiers and mixed block modes,

c) Overlap add of IMDCT outputs, and

d) Frequency inversion prior to pseudo-quadrature mirror synthesis filter (PQMF) bank.

Because the IMDCT is a lapped transform, the user must preallocate a buffer referenced by *pSrcDstOverlapAdd* to maintain the IMDCT overlap-add state. The buffer must contain 576 elements. Prior to the first call to the synthesis filter bank, all elements of the overlap-add buffer should be set equal to zero. In between all subsequent calls, the MP3 application must preserve the contents of the overlap-add buffer. Upon entry to `ippsMDCTInv_MP3_32s`, the overlap-add buffer should contain the IMDCT output generated by operating on the previous granule; upon exit from `ippsMDCTInv_MP3_32s`, the overlap-add buffer will contain the overlapped portion of the output generated by operating on the current granule.
Upon return from the function, the IMDCT sub-band output samples are organized as follows:

pDstY`[j*32+subband]`, for `j`=0 to 17; `subband`=0 to 31.

Note that the pointers *pSrcXr* and *pDstY* must reference different buffers.

Q5.26 designates that 5 bits before and 26 bits after fixed point position are used to present a 32-bit value in the fixed point format.
Q7.24 designates that 7 bits before and 24 bits after fixed point position are used to present a 32-bit value in the fixed point format.

Return Value

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsErr	Indicates an error when one or more of the following input data errors are detected: either *blockType* exceeds [0,3], or *mixedBlock* exceeds [0,1], or *nonZeroBound* exceeds [0,576], or **pPrevNumOfImdct* exceeds [0,32].

SynthPQMF_MP3

Performs the second stage of hybrid synthesis filter bank.

```
IppStatus ippsSynthPQMF_MP3_32s16s(Ipp32s* pSrcY,
     Ipp16s* pDstAudioOut, Ipp32s* pVBuffer, int* pVPosition,
     int mode);
```

Arguments

pSrcY	Pointer to the block of 32 IMDCT sub-band input samples, in Q7.24 format.
pDstAudioOut	Pointer to a block of 32 reconstructed PCM output samples in 16-bit signed format (little-endian); left and right channels are interleaved according to the *mode* flag.

pVBuffer	Pointer to the input workspace buffer containing Q7.24 data; it will be updated in the function.
pVPosition	Pointer to the internal workspace index; it will be updated in the function.
mode	Flag that indicates whether or not the PCM audio output channels should be interleaved (1- not interleaved, 2 - interleaved).

Discussion

The function ippsSynthPQMF_MP3 performs the second stage of the hybrid synthesis filter bank: a critically-sampled 32-channel PQMF synthesis bank that generates 32 time-domain output samples for each 32-sample input block of IMDCT outputs. For each input block, the PQMF generates an output sequence of 16-bit signed little-endian PCM samples in the vector pointed to by *pDstAudioOut*.
If *mode* equals 2, the left and right channel output samples are interleaved (that is, LRLRLR), such that the left channel data is organized as follows:
pDstAudioOut [2*i], i=0 to 31.
If *mode* equals 1, then the left and right channel outputs are not interleaved.

Because the PQMF bank contains memory, the MP3 application must maintain two state variables in between calls to the function. First, the application must preallocate a workspace buffer of size 512 x (Number of Channels) for the PQMF computation. This buffer is referenced by the pointer *pVBuffer*, and its elements should be initialized to zero prior to the first call. During subsequent calls, the *pVBuffer* input for the current call should contain the *pVbuffer* output generated by the previous call. In addition to *pVBuffer*, the MP3 application must also initialize to zero and thereafter preserve the value of the state variable *pVPosition*. The MP3 application should modify the values contained in *pVBuffer* or *pVPosition* only during decoder reset, and the reset values should always be zero.

Return Value

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error condition if at least one of the specified pointers is NULL, or the value of *mode* is not equal to 1 or 2.
ippStsErr	Indicates an error condition if the value of *pVPosition* exceeds [0, 15].

Fixed-Accuracy Arithmetic Functions

11

This chapter describes IPP fixed-accuracy transcendental mathematical functions of vector arguments. These functions take an input vector as argument, compute values of the respective elementary function element-wise, and return the results in an output vector.

Function specifications comply with the common API agreement of IPP, but include some new features essential to scientific arithmetic functions. The main feature is a more elaborate specification of accuracy that differs from the common definition in adding several new levels of accuracy, besides original levels introduced by single precision and double precision data formats.

Fixed-accuracy vector functions implementation supports IEEE-754 standard in all flavors, which means that:

- All functions have a precisely determined and guaranteed level of accuracy for all argument values.
- All special value processing and exceptions handling requirements are met, which implies that when accuracy is below the standard level, the function meets IEEE-754 requirements in all other respects.

The choice of accuracy levels should be based on practical experience and identified application demands. Available options are specified in the function name suffix and include `A11`, `A21`, or `A24` for the single precision, and `A50` or `A53` for the double precision data format. Flavors `A11`, `A21`, and `A50` provide approximately 3, 6, and 15 exact decimal digits, respectively. For flavors `A24` and `A53`, the maximum guaranteed error is within 1 ulp and in most cases does not exceed 0.55 ulp.

Fixed-accuracy arithmetic functions subset of IPP has the similar functionality as the respective part of Intel Math Kernel Library (MKL) . However, IPP provides lower-level transcendental functions that have separate flavors for each mode of operations and data type and are better suitable for multimedia and signal processing in real time applications.

The full list of these functions is given in Table 11-1.

Table 11-1 IPP Fixed-Accuracy Arithmetic Functions

Function Short Name	Data Types	Description
Power and Root Functions		
Inv	32f, 64f	Computes inverse value of each vector element.
Div	32f, 64f	Divides elements of one vector by corresponding elements of another vector.
Sqrt	32f, 64f	Computes square root of each vector element.
InvSqrt	32f, 64f	Computes inverse square root of each vector element.
Cbrt	32f, 64f	Computes cube root of each vector element.
InvCbrt	32f, 64f	Computes inverse cube root of each vector element.
Pow	32f, 64f	Raises each element of one vector to the power of corresponding element of another vector.
Exponential and Logarithmic Functions		
Exp	32f, 64f	Raises *e* to the power of each vector element.
Ln	32f, 64f	Computes natural logarithm of each vector element.
Log10	32f, 64f	Computes common logarithm of each vector element.
Trigonometric Functions		
Cos	32f, 64f	Computes cosine of each vector element.
Sin	32f, 64f	Computes sine of each vector element.
SinCos	32f, 64f	Computes sine and cosine of each vector element.
Tan	32f, 64f	Computes tangent of each vector element.
Acos	32f, 64f	Computes inverse cosine of each vector element.
Asin	32f, 64f	Computes inverse sine of each vector element.
Atan	32f, 64f	Computes inverse tangent of each vector element.

Table 11-1 IPP Fixed-Accuracy Arithmetic Functions (continued)

Function Short Name	Data Types	Description
Atan2	32f, 64f	Computes four-quadrant inverse tangent of elements of two vectors.
Hyperbolic Functions		
Cosh	32f, 64f	Computes hyperbolic cosine of each vector element.
Sinh	32f, 64f	Computes hyperbolic sine of each vector element.
Tanh	32f, 64f	Computes hyperbolic tangent of each vector element.
Acosh	32f, 64f	Computes inverse (nonnegative) hyperbolic cosine of each vector element.
Asinh	32f, 64f	Computes inverse hyperbolic sine of each vector element.
Atanh	32f, 64f	Computes inverse hyperbolic tangent of each vector element.

NOTE. *You should not confuse fixed-accuracy arithmetic functions described in this chapter with functions in chapter 5 that have similar functionality but follow different accuracy specifications.*

IPP fixed-accuracy arithmetic functions may return status codes of normal execution (`IppStsNoErr`), error conditions `IppStsSizeErr`, `IppStsNullPtrErr` (see Table 2-2 in Chapter 2), and the following specific warnings: `IppStsDomainErr`, `IppStsSingErr`, `IppStsOverflowErr`, `IppStsUnderflowErr`. In case of warnings, the value returned is positive and the computation is continued.

Table 11-2 lists status codes and the corresponding messages for these warnings.

Table 11-2 Warning Status Codes for Fixed-Accuracy Arithmetic Functions

Status	Value	Message
`IppStsDomainErr`	1	Domain error
`IppStsSingErr`	2	Singularity error
`IppStsOverflowErr`	3	Overflow error
`IppStsUnderflowErr`	4	Underflow error

Power and Root Functions

Inv

Computes inverse value of each vector element.

```
IppStatus ippsInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsInv computes the inverse value of each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor ippsInv_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsInv_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsInv_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsInv_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsInv_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \frac{1}{pSrc[n]}, 0 \le n < len .$$

Example 11-1 shows how to use the function ippsInv.

Example 11-1 Using ippsInv Function

```
IppStatus ippsInv_32f_A21_sample(void)
{
const Ipp32fx[4] = {-9.975, 1.272, -6.134, 6.175};
Ipp32f      y[4];

    IppStatus st = ippsInv_32f_A21( x, y, 4 );

    printf(" ippsInv_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsInv_32f_A21:
 x = -9.975 1.272 -6.134 6.175
 y = -0.100 0.786 -0.163 0.162
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsSingErr	Indicates an error when argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

Div

Divides each element of the first vector by corresponding element of the second vector.

```
IppStatus ippsDiv_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsDiv_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f*  pDst, int len);
IppStatus ippsDiv_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f*  pDst, int len);
IppStatus ippsDiv_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
IppStatus ippsDiv_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
```

Arguments

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsDiv divides each element of the vector *pSrc1* by the corresponding element of the vector *pSrc2* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsDiv_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsDiv_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsDiv_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsDiv_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsDiv_64f_A53 guarantees 53 correctly rounded bits of significand, including the impede bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as $pDst[n] = \frac{pSrc1[n]}{pSrc2[n]}, 0 \le n < len$.

Example 11-2 shows how to use the function ippsDiv.

Example 11-2 Using ippsDiv Function

```
IppStatus ippsDiv_32f_A21_sample(void)
{
const Ipp32fx1[4] = {599.088, 735.034, 572.448, 151.640};
const Ipp32fx2[4] = {385.297, 609.005, 361.403, 225.182};
Ipp32f      y[4];

    IppStatus st = ippsDiv_32f_A21( x1, x2, y, 4 );

    printf(" ippsDiv_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}

Output results:

 ippsDiv_32f_A21:
 x1 = 599.088 735.034 572.448 151.640
 x2 = 385.297 609.005 361.403 225.182
 y  = 1.555 1.207 1.584 0.673
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc1* or *pSrc2* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsSingErr	Indicates an error when argument is the singularity point, that is, at least one of the elements of *pSrc2* is equal to 0.

Sqrt

Computes square root of each vector element.

```
IppStatus ippsSqrt_32f_A11 ( const Ipp32f*  pSrc, Ipp32f*  pDst, int len );
IppStatus ippsSqrt_32f_A21 ( const Ipp32f*  pSrc, Ipp32f*  pDst, int len );
IppStatus ippsSqrt_32f_A24 ( const Ipp32f*  pSrc, Ipp32f*  pDst, int len );
IppStatus ippsSqrt_64f_A50 ( const Ipp64f*  pSrc, Ipp64f*  pDst, int len );
IppStatus ippsSqrt_64f_A53 ( const Ipp64f*  pSrc, Ipp64f*  pDst, int len );
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsSqrt computes square root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsSqrt_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsSqrt_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsSqrt_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsSqrt_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsSqrt_64f_A53 guarantees 53 correctly rounded bits of significand, including the impede bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \sqrt{pSrc[n]}, 0 \le n < len.$

Example 11-3 shows how to use the function ippsSqrt.

Example 11-3 Using ippsSqrt Function

```
IppStatus ippsSqrt_32f_A21_sample(void)
{
const Ipp32fx[4] = {5850.093, 4798.730, 3502.915, 8959.624};
Ipp32f      y[4];

    IppStatus st = ippsSqrt_32f_A21( x, y, 4 );

    printf(" ippsSqrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsSqrt_32f_A21:
 x = 5850.093 4798.730 3502.915 8959.624
 y = 76.486 69.273 59.185 94.655
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDomainErr	Indicates an error when argument is out of function domain, that is, at least one of the elements of *pSrc* is less than 0.

InvSqrt

Computes inverse square root of each vector element.

```
IppStatus ippsInvSqrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsInvSqrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsInvSqrt computes inverse square root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsInvSqrt_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsInvSqrt_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsInvSqrt_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsInvSqrt_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsInvSqrt_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \frac{1}{\sqrt{pSrc[n]}}, 0 \le n < len.$$

Example 11-4 shows how to use the function ippsInvSqrt.

Example 11-4 Using ippsInvSqrt Function

```
 IppStatus ippsInvSqrt_32f_A21_sample(void)
{
const Ipp32fx[4] = {7105.043, 5135.398, 3040.018, 149.944};
Ipp32f      y[4];

    IppStatus st = ippsInvSqrt_32f_A21( x, y, 4 );

    printf(" ippsInvSqrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsInvSqrt_32f_A21:
 x = 7105.043 5135.398 3040.018 149.944
 y = 0.012 0.014 0.018 0.082
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDomainErr	Indicates an error when argument is out of function domain, that is, at least one of the elements of *pSrc* is less than 0.
ippStsSingErr	Indicates an error when argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

Cbrt

Computes cube root of each vector element.

```
IppStatus ippsCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsCbrt computes cube root of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsCbrt_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsCbrt_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsCbrt_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsCbrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsCbrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \sqrt[3]{pSrc[n]}, 0 \le n < len$.

Example 11-5 shows how to use the function `ippsCbrt`.

Example 11-5 Using `ippsCbrt` Function

```
IppStatus ippsCbrt_32f_A21_sample(void)
{
const Ipp32fx[4] = {6456.801, 4932.096, -6517.838, 7178.869};
Ipp32f      y[4];

    IppStatus st = ippsCbrt_32f_A21( x, y, 4 );

    printf(" ippsCbrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsCbrt_32f_A21:
 x = 6456.801 4932.096 -6517.838 7178.869
 y = 18.621 17.022 -18.680 19.291
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`* or *`pDst`* pointer is null.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.

InvCbrt

Computes inverse cube root of each vector element.

```
IppStatus ippsInvCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsInvCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function `ippsInvCbrt` computes inverse cube root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsInvCbrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsInvCbrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsInvCbrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsInvCbrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsInvCbrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \frac{1}{\sqrt[3]{pSrc[n]}}, 0 \le n < len.$$

Example 11-6 shows how to use the function `ippsInvCbrt`.

Example 11-6 Using `ippsInvCbrt` Function

```
IppStatus ippsInvCbrt_32f_A21_sample(void)
{
const Ipp32fx[4] = {914.120, 3644.584, 1473.214, 1659.070};
Ipp32f      y[4];

    IppStatus st = ippsInvCbrt_32f_A21( x, y, 4 );

    printf(" ippsInvCbrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsInvCbrt_32f_A21:
 x = 914.120 3644.584 1473.214 1659.070
 y = 0.103 0.065 0.088 0.084
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`* or *`pDst`* pointer is null.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`ippStsSingErr`	Indicates an error when argument is the singularity point, that is, at least one of the elements of *`pSrc`* is equal to 0.

Pow

Raises each element of the first vector to the power of corresponding element of the second vector.

```
IppStatus ippsPow_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
IppStatus ippsPow_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
```

Arguments

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsPow raises each element of vector *pSrc1* to the power of the corresponding element of the vector *pSrc2* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsPow_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsPow_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsPow_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsPow_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsPow_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = (pSrc1[n])^{pSrc2[n]}, 0 \le n < len.$

Example 11-7 shows how to use the function ippsPow.

Example 11-7 Using ippsPow Function

```
IppStatus ippsPow_32f_A21_sample(void)
{
const Ipp32fx1[4] = {0.483, 0.565, 0.776, 0.252};
const Ipp32fx2[4] = {0.823, 0.991, 0.411, 0.692};
Ipp32f      y[4];

    IppStatus st = ippsPow_32f_A21( x1, x2, y, 4 );

    printf(" ippsPow_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}


Output results:

 ippsPow_32f_A21:
 x1 = 0.483 0.565 0.776 0.252
 x2 = 0.823 0.991 0.411 0.692
 y  = 0.549 0.568 0.901 0.386
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc1`* or *`pSrc2`* or *`pDst`* pointer is null.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`ippStsDomainErr`	Indicates an error when argument is out of function domain, that is, at least one pair of the source elements meets the following condition: element of *`pSrc1`* is finite, less than 0, and element of *`pSrc2`* is finite, non-integer.
`ippStsSingErr`	Indicates an error when argument is the singularity point, that is, at least one pair of the elements is as follows: element of *`pSrc1`* is equal to 0, and element of *`pSrc2`* is integer and less than 0.

Exponential and Logarithmic Functions

Exp

Raises e to the power of each vector element.

```
IppStatus ippsExp_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsExp raises *e* to the power of each element of *pSrc* and stores the result in the corresponding element of *pDst*.
For single precision data:

function flavor ippsExp_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsExp_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsExp_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsExp_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsExp_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = e^{pSrc[n]}, 0 \le n < len.$

Example 11-8 shows how to use the function ippsExp.

Example 11-8 Using ippsExp Function

```
IppStatus ippsExp_32f_A21_sample(void)
{
const Ipp32fx[4] = {4.885, -0.543, -3.809, -4.953};
Ipp32f      y[4];

    IppStatus st = ippsExp_32f_A21( x, y, 4 );

    printf(" ippsExp_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsExp_32f_A21:
 x = 4.885 -0.543 -3.809 -4.953
 y = 132.324 0.581 0.022 0.007
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsOverflowErr	Indicates an error when function overflows, that is, at least one of elements of *pSrc* is greater then Ln(FPMAX), where FPMAX is the maximum representable floating-point number.
IppStsUnderflowErr	Indicates an error when function underflows, that is, at least one of elements of *pSrc* is less then Ln(FPMIN), where FPMIN is the minimum positive floating-point value.

Ln

Computes natural logarithm of each vector element.

```
IppStatus ippsLn_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLn_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsLn computes a natural logarithm of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsLn_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsLn_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsLn_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsLn_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsLn_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \log_e(pSrc[n]), 0 \le n < len.$

Example 11-9 shows how to use the function ippsLn.

Example 11-9 Using ippsLn Function

```
IppStatus ippsLn_32f_A21_sample(void)
{
const Ipp32fx[4] = {0.188, 3.841, 5.363, 5.755};
Ipp32f      y[4];

    IppStatus st = ippsLn_32f_A21( x, y, 4 );

    printf(" ippsLn_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}


Output results:

 ippsLn_32f_A21:
 x = 0.188 3.841 5.363 5.755
 y = -1.670 1.346 1.680 1.750
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDomainErr	Indicates an error when argument is out of function domain, that is, at least one of the elements of *pSrc* is less than 0.
ippStsSingErr	Indicates an error when argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

Log10

Computes common logarithm of each vector element.

```
IppStatus ippsLog10_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLog10_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsLog10 computes a natural logarithm of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsLog10_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsLog10_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsLog10_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsLog10_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsLog10_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \log_{10}(pSrc[n]), 0 \le n < len.$

Example 11-10 shows how to use the function ippsLog10.

Example 11-10 Using ippsLog10 Function

```
IppStatus ippsLog10_32f_A21_sample(void)
{
const Ipp32fx[4] = {6.057, 6.111, 1.746, 6.664};
Ipp32f      y[4];

    IppStatus st = ippsLog10_32f_A21( x, y, 4 );

    printf(" ippsLog10_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsLog10_32f_A21:
 x = 6.057 6.111 1.746 6.664
 y = 0.782 0.786 0.242 0.824
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDomainErr	Indicates an error when argument is out of function domain, that is, at least one of the elements of *pSrc* is less than 0.
ippStsSingErr	Indicates an error when argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

Trigonometric Functions

Cos

Computes cosine of each vector element.

```
IppStatus ippsCos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsCos computes a cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsCos_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsCos_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsCos_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsCos_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsCos_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \cos(pSrc[n]), 0 \le n < len$.

Example 11-11 shows how to use the function ippsCos.

Example 11-11 Using ippsCos Function

```
IppStatus ippsCos_32f_A21_sample(void)
{
const Ipp32fx[4] = {-984.222, -2957.549, -8859.218, 2153.691};
Ipp32f      y[4];

    IppStatus st = ippsCos_32f_A21( x, y, 4 );

    printf(" ippsCos_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsCos_32f_A21:
 x = -984.222 -2957.549 -8859.218 2153.691
 y = -0.619 -0.258 0.997 0.129
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDomainErr	Indicates an error when argument is out of function domain, that is, at least one of the elements of *pSrc* is equal to ± INF.

Sin

Computes sine of each vector element.

```
IppStatus ippsSin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsSin computes a sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsSin_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsSin_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsSin_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsSin_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsSin_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \sin(pSrc[n]), 0 \le n < len$.

Example 11-12 shows how to use the function ippsSin.

Example 11-12 Using ippsSin Function

```
IppStatus ippsSin_32f_A21_sample(void)
{
const Ipp32fx[4] = {5666.372, 6052.125, 397.656, -3960.997};
Ipp32f      y[4];

    IppStatus st = ippsSin_32f_A21( x, y, 4 );

    printf(" ippsSin_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsSin_32f_A21:
 x = 5666.372 6052.125 397.656 -3960.997
 y = -0.873 0.988 0.970 -0.524
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDomainErr	Indicates an error when argument is out of function domain, that is, at least one of the elements of *pSrc* is equal to ± INF.

SinCos

Computes sine and cosine of each vector element.

```
IppStatus ippsSinCos_32f_A11 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_32f_A21 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_32f_A24 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_64f_A50 (const Ipp64f* pSrc, const Ipp64f* pDst1, Ipp64f*
    pDst2, int len);
IppStatus ippsSinCos_64f_A53 (const Ipp64f* pSrc, const Ipp64f* pDst1, Ipp64f*
    pDst2, int len);
```

Arguments

pSrc	Pointer to the first source vector.
pDst1	Pointer to the destination vector for sine values.
pDst2	Pointer to the destination vector for cosine values.
len	Number of elements in the vectors.

Discussion

The function `ippsSinCos` computes sine of each element of *pSrc* and stores the result in the corresponding element of *pDst1*; computes cosine of each element of *pSrc* and stores the result in the corresponding element of *pDst2*.

For single precision data:

function flavor `ippsSinCos_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsSinCos_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsSinCos_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsSinCos_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsSinCos_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst1[n] = \sin(pSrc[n]), pDst2[n] = \cos(pSrc[n]), 0 \le n < len.$

Example 11-13 shows how to use the function ippsSinCos.

Example 11-13 Using ippsSinCos Function

```
IppStatus ippsSinCos_32f_A21_sample(void)
{
const Ipp32fx[4] = {3857.845, -3939.024, -1468.856, -8592.486};
Ipp32f      y1[4];
Ipp32f      y2[4];

    IppStatus st = ippsSinCos_32f_A21( x, y1, y2, 4 );

    printf(" ippsSinCos_32f_A21:\n");
    printf(" x  = %.3f %.3f %.3f %.3f \n", x[0],  x[1],  x[2],  x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
    return st;
}

Output results:

 ippsSinCos_32f_A21:
 x  = 3857.845 -3939.024 -1468.856 -8592.486
 y1 = -0.031 0.508 0.987 0.228
 y2 = 1.000 0.861 0.161 -0.974
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDst1* or *pDst2* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDomainErr	Indicates an error when argument is out of function domain, that is, at least one of the *pSrc* elements is equal to ± INF.

Tan

Computes tangent of each vector element.

```
IppStatus ippsTan_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsTan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsTan computes the tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsTan_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsTan_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsTan_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsTan_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsTan_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows: $pDst[n] = \tan(pSrc[n]), 0 \le n < len$.

Example 11-14 shows how to use the function ippsTan.

Example 11-14 Using ippsTan Function

```
IppStatus ippsTan_32f_A21_sample(void)
{
const Ipp32fx[4] = {7519.456, 4533.524, 9118.015, 8514.359};
Ipp32f      y[4];

    IppStatus st = ippsTan_32f_A21( x, y, 4 );

    printf(" ippsTan_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsTan_32f_A21:
 x = 7519.456 4533.524 9118.015 8514.359
 y = -18.656 0.209 2.028 0.750
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDomainErr	Indicates an error when argument is out of function domain, that is, at least one of the elements of *pSrc* is equal to ± INF.

Acos

Computes inverse cosine of each vector element.

```
IppStatus ippsAcos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAcos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAcos computes the inverse cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsAcos_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAcos_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAcos_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsAcos_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsAcos_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{acos}(pSrc[n]), 0 \le n < len.$

Example 11-15 shows how to use the function ippsAcos.

Example 11-15 Using ippsAcos Function

```
IppStatus ippsAcos_32f_A21_sample(void)
{
const Ipp32fx[4] = {0.079, -0.715, -0.076, -0.529};
Ipp32f      y[4];

    IppStatus st = ippsAcos_32f_A21( x, y, 4 );

    printf(" ippsAcos_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAcos_32f_A21:
 x = 0.079 -0.715 -0.076 -0.529
 y = 1.492 2.368 1.647 2.129
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDomainErr	Indicates an error when argument is out of function domain, that is, at least one of the elements of *pSrc* has an absolute value greater than 1.

Asin

Computes inverse sine of each vector element.

```
IppStatus ippsAsin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAsin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAsin computes the inverse sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsAsin_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAsin_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAsin_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsAsin_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsAsin_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows: $pDst[n] = \sin(pSrc[n]), 0 \le n < len$.

Example 11-16 shows how to use the function ippsAsin.

Example 11-16 Using ippsAsin Function

```
IppStatus ippsAsin_32f_A21_sample(void)
{
const Ipp32fx[4] = {0.724, -0.581, 0.559, 0.687};
Ipp32f      y[4];

    IppStatus st = ippsAsin_32f_A21( x, y, 4 );

    printf(" ippsAsin_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}


Output results:

 ippsAsin_32f_A21:
 x = 0.724 -0.581 0.559 0.687
 y = 0.810 -0.620 0.594 0.758
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDomainErr	Indicates an error when argument is out of function domain, that is, at least one of the elements of *pSrc* has an absolute value greater than 1.

Atan

Computes inverse tangent of each vector element.

```
IppStatus ippsAtan_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAtan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAtan computes the inverse tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsAtan_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAtan_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAtan_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsAtan_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsAtan_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{atan}(pSrc[n]), 0 \le n < len.$

Example 11-17 shows how to use the function ippsAtan.

Example 11-17 Using ippsAtan Function

```
IppStatus ippsAtan_32f_A21_sample(void)
{
const Ipp32fx[4] = {0.994, 0.999, 0.223, -0.215};
Ipp32f      y[4];

    IppStatus st = ippsAtan_32f_A21( x, y, 4 );

    printf(" ippsAtan_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAtan_32f_A21:
 x = 0.994 0.999 0.223 -0.215
 y = 0.782 0.785 0.219 -0.212
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Atan2

Computes four-quadrant inverse tangent of elements of two vectors.

```
IppStatus ippsAtan2_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsAtan2_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsAtan2_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsAtan2_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int len);
IppStatus ippsAtan2_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int len);
```

Arguments

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAtan2 computes the angle between the X axis and the line from the origin to the point (X,Y), for each element of *pSrc1* as a Y (the ordinate) and corresponding element of *pSrc2* as an X (the abscissa), and stores the result in the corresponding element of *pDst*. The result angle varies from - π to + π.

For single precision data:

function flavor ippsAtan2_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAtan2_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAtan2_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsAtan2_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsAtan2_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = arctan2(pSrc1[n], pSrc2[n]) \quad , 0 \le n < len .$

Example 11-18 shows how to use the function ippsAtan2.

Example 11-18 Using ippsAtan2 Function

```
IppStatus ippsAtan2_32f_A21_sample(void)
{
const Ipp32fx1[4] = {1.492, 1.700, 1.147, 1.142};
const Ipp32fx2[4] = {1.064, 1.505, 1.950, 1.905};
Ipp32f      y[4];

    IppStatus st = ippsAtan2_32f_A21( x1, x2, y, 4 );

    printf(" ippsAtan2_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}


Output results:

 ippsAtan2_32f_A21:
 x1 = 1.492 1.700 1.147 1.142
 x2 = 1.064 1.505 1.950 1.905
 y  = 0.951 0.846 0.532 0.540
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc1`* or *`pSrc2`* or *`pDst`* pointer is null.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.

Hyperbolic Functions

Cosh

Computes hyperbolic cosine of each vector element.

```
IppStatus ippsCosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsCosh computes the hyperbolic cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsCosh_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsCosh_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsCosh_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsCosh_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsCosh_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \cosh(pSrc[n]), 0 \le n < len.$

Example 11-19 shows how to use the function ippsCosh.

Example 11-19 Using ippsCosh Function

```
IppStatus ippsCosh_32f_A21_sample(void)
{
const Ipp32fx[4] = {-4.676, -4.054, 6.803, -9.525};
Ipp32f      y[4];

    IppStatus st = ippsCosh_32f_A21( x, y, 4 );

    printf(" ippsCosh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}


Output results:

 ippsCosh_32f_A21:
 x = -4.676 -4.054 6.803 -9.525
 y = 53.661 28.833 450.219 6849.870
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsOverflowErr	Indicates an error when the function overflows, that is, at least one of elements of *pSrc* has the absolute value greater then Ln(FPMAX) + Ln(2), where FPMAX is the maximum representable floating-point number.

Sinh

Computes hyperbolic sine of each vector element.

```
IppStatus ippsSinh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsSinh computes the hyperbolic sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsSinh_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsSinh_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsSinh_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsSinh_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsSinh_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \sinh(pSrc[n]), 0 \le n < len$.

Example 11-20 shows how to use the function ippsSinh.

Example 11-20 Using ippsSinh Function

```
IppStatus ippsSinh_32f_A21_sample(void)
{
const Ipp32fx[4] = {-2.483, -8.148, 3.544, -8.876};
Ipp32f      y[4];

    IppStatus st = ippsSinh_32f_A21( x, y, 4 );

    printf(" ippsSinh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsSinh_32f_A21:
 x = -2.483 -8.148 3.544 -8.876
 y = -5.945 -1727.412 17.290 -3577.970
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsOverflowErr	Indicates an error when the function overflows, that is, at least one of elements of *pSrc* has the absolute value greater then Ln(FPMAX) + Ln(2), where FPMAX is the maximum representable floating-point number.

Tanh

Computes hyperbolic tangent of each vector element.

```
IppStatus ippsTanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsTanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsTanh computes the hyperbolic tangent of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsTanh_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsTanh_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsTanh_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsTanh_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsTanh_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[x] = \tanh(pSrc[n]), 0 \le n < len.$

Example 11-21 shows how to use the function ippsTanh.

Example 11-21 Using ippsTanh Function

```
IppStatus ippsTanh_32f_A21_sample(void)
{
const Ipp32fx[4] = {-0.982, 0.838, -0.448, -0.454};
Ipp32f      y[4];

    IppStatus st = ippsTanh_32f_A21( x, y, 4 );

    printf(" ippsTanh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}


Output results:

 ippsTanh_32f_A21:
 x = -0.982 0.838 -0.448 -0.454
 y = -0.754 0.685 -0.420 -0.425
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Acosh

Computes inverse (nonnegative) hyperbolic cosine of each vector element.

```
IppStatus ippsAcosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAcosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAcosh computes the inverse (nonnegative) hyperbolic cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsAcosh_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAcosh_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAcosh_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsAcosh_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsAcosh_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[x] = \text{acosh}(pSrc[n]), 0 \le n < len.$

Example 11-22 shows how to use the function ippsAcosh.

Example 11-22 Using ippsAcosh Function

```
IppStatus ippsAcosh_32f_A21_sample(void)
{
const Ipp32fx[4] = {588.321, 691.492, 837.773, 726.767};
Ipp32f      y[4];

    IppStatus st = ippsAcosh_32f_A21( x, y, 4 );

    printf(" ippsAcosh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAcosh_32f_A21:
 x = 588.321 691.492 837.773 726.767
 y = 7.070 7.232 7.424 7.282
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDomainErr	Indicates an error when argument is out of function domain, that is, at least one of the elements of *pSrc* is less than 1.

Asinh

Computes inverse hyperbolic sine of each vector element.

```
IppStatus ippsAsinh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAsinh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAsinh computes the inverse hyperbolic sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsAsinh_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAsinh_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAsinh_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsAsinh_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsAsinh_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{asinh}(pSrc[n]), 0 \le n < len.$

Example 11-23 shows how to use the function `ippsAsinh`.

Example 11-23 Using `ippsAsinh` Function

```
IppStatus ippsAsinh_32f_A21_sample(void)
{
const Ipp32fx[4] = {-30.122, -589.282, 487.472, -63.082};
Ipp32f      y[4];

    IppStatus st = ippsAsinh_32f_A21( x, y, 4 );

    printf(" ippsAsinh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAsinh_32f_A21:
 x = -30.122 -589.282 487.472 -63.082
 y = -4.099 -7.072 6.882 -4.838
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`* or *`pDst`* pointer is null.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.

Atanh

Computes inverse hyperbolic tangent of each vector element.

```
IppStatus ippsAtanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAtanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAtanh computes the inverse hyperbolic tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsAtanh_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAtanh_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAtanh_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsAtanh_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsAtanh_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{atanh}(pSrc[n]), 0 \le n < len.$

Example 11-24 shows how to use the function `ippsAtanh`.

Example 11-24 Using `ippsAtanh` Function

```
IppStatus ippsAtanh_32f_A21_sample(void)
{
const Ipp32fx[4] = {-0.076, 0.808, 0.440, -0.705};
Ipp32f      y[4];

    IppStatus st = ippsAtanh_32f_A21( x, y, 4 );

    printf(" ippsAtanh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAtanh_32f_A21:
 x = -0.076 0.808 0.440 -0.705
 y = -0.076 1.123 0.472 -0.877
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`* or *`pDst`* pointer is null.
`ippStsSizeErr`	Indicates an error when len is less than or equal to 0.
`ippStsDomainErr`	Indicates an error when argument is out of function domain, that is, at least one of the elements of *`pSrc`* has absolute value greater than 1.
`ippStsSingErr`	Indicates an error when argument is the singularity point, that is, at least one of the elements of *`pSrc`* has absolute value equal to 1.

Bibliography

This bibliography provides a list of reference books that might be useful to the application programmer. This list is neither complete nor exhaustive, but serves as a starting point. Of all the references listed, [Mit93] will be the most useful to those readers who already have a basic understanding of signal processing. This reference collects the work of 27 experts in the field and has both great breadth and depth.

The books [Opp75], [Opp89], [Jac89], and [Zie83] are undergraduate signal processing texts. [Opp89] is a much revised edition of the classic [Opp75]; [Jac89] is more concise than the others; and [Zie83] also covers continuous-time systems.

[Bri94] C. Brislawn, *Classification of Nonexpansive Symmetric Extension Transforms for Multirate Filter Banks*. Los Alamos Report LA-UR-94-1747, 1994.

[Cap78] V. Cappellini, A. G. Constantinides, and P. Emilani. *Digital Filters and Their Applications*. Academic Press, London, 1978.

[CCITT] CCITT, Recommendation G.711

[Cro83] R. E. Crochiere and L. R. Rabiner. *Multirate Digital Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1983.

[Dau92] I. Daubechies. *Ten Lectures on Wavelets*. Springer Verlag, Pennsylvania, 1992.

[Fei92] E. Feig and S. Winograd. *Fast algorithms for DCT*. IEEE Transactions on Signal Processing, vol.40, No.9, 1992.

[Har78] F. Harris. *On the Use of Windows*. Proceedings of the IEEE, vol. 66, No.1, IEEE, 1978.

[Hay91] S. Haykin. *Adaptive Filter Theory*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Jac89] Leland B. Jackson. *Digital Filters and Signal Processing*. Kluwer Academic Publishers, second edition, 1989.

[Lyn89] Paul A. Lynn. *Introductory Digital Signal Processing with Computer Applications*. John Wiley&Sons, Inc., New York, 1993.

[Mit93] Sanjit K. Mitra and James F. Kaiser editors. *Handbook for Digital Signal Processing*. John Wiley&Sons, Inc., New York, 1993.

[Mit98] S. K. Mitra. *Digital Signal Processing*. McGraw Hill, 1998.

[NIC91] Nam Ik Cho and Sang Uk Lee. *Fast algorithm and implementation of 2D DCT.* IEEE Transactions on Circuits and Systems, vol. 31, No.3, 1991.

[Opp75] Alan V. Oppenheim and Ronald W. Schafer. *Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[Opp89] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Rab78] L.R. Rabiner and R.W. Schafer. *Digital Processing of Speech Signals*. Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[Rao90] K.R. Rao and P. Yip. *Discrete Cosine Transform. Algorithms, Advantages and Applications*. Academic Press, San Diego, 1990.

[Str96] G. Strang and T. Nguyen. *Wavelet and Filter Banks*. Wellesley-Cambridge Press, 1996.

[Vai93] P. P. Vaidyanathan. *Multirate Systems and Filter Banks*. Prentice Hall, Englewood Cliffs, New Jersey.

[Wid85] B. Widrow and S.D. Stearns. *Adaptive Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[Zie83] Rodger E. Ziemer, William H. Tranter and D. Ronald Fannin. *Signals and Systems: Continuous and Discrete*. Macmillan Publishing Co., New York, 1983.

Glossary

adaptive filter	An adaptive filter varies its filter coefficients (taps) over time. Typically, the filter's coefficients are varied to make its output match a prototype "desired" signal as closely as possible. Non-adaptive filters do not vary their filter coefficients over time.
arithmetic operation	An operation that adds, subtracts, multiplies, or squares the image pixel values.
BQ	One of the modes, which indicates that the IIR initialization function initializes a cascade of biquads.
CCS	See complex conjugate-symmetric.
companding functions	The functions that perform an operation of data compression by using a logarithmic encoder-decoder. Companding allows you to maintain the percentage error constant by logarithmically spacing the quantization levels.
complex conjugate-symmetric	A kind of symmetry that arises in the Fourier transform of real signals. A complex conjugate-symmetric signal has the property that `x(-n) = x(n)*`, where "`*`" denotes conjugation.
conjugate	The conjugate of a complex number `a + bj` is `a - bj`.
conjugate-symmetric	See complex conjugate-symmetric.
DCT	Acronym for the discrete cosine transform.
decimation	Filtering a signal followed by down-sampling. Filtering prevents aliasing distortion in the subsequent down-sampling. See down-sampling.

down-sampling	Down-sampling conceptually decreases a signal's sampling rate by removing samples from between neighboring samples of a signal. See decimation.
element-wise	An element-wise operation performs the same operation on each element of a vector, or uses the elements of the same position in multiple vectors as inputs to the operation. For example, the element-wise addition of the vectors $\{x_0, x_1, x_2\}$ and $\{y_0, y_1, y_2\}$ is performed as follows: $\{x_0, x_1, x_2\} + \{y_0, y_1, y_2\} = \{x_0 + y_0, x_1 + y_1, x_2 + y_2\}$.
FIR	Abbreviation for finite impulse response filter. Finite impulse response filters do not vary their filter coefficients (taps) over time.
FIR LMS	Abbreviation for least mean squares finite impulse response filter.
fixed-point data format	A format that assigns one bit for a sign and all other bits for fractional part. This format is used for optimized conversion operations with signed, purely fractional vectors. For example, S.31 format assumes a sign bit and 31 fractional bits; S15.16 assumes a sign bit, 15 integer bits, and 16 fractional bits.
IIR	Abbreviation for infinite impulse response filters.
in-place	A function that performs its operation in-place, takes its input from an array and returns its output to the same array. See not-in-place.
interpolation	Up-sampling a signal followed by filtering. The filtering gives the inserted samples a value close to the samples of their neighboring samples in the original signal. See up-sampling.
LMS	Abbreviation for least mean square, an algorithm frequently used as a measure of the difference between two signals. Also used as shorthand for an adaptive FIR filter employing the LMS algorithm for adaptation.

LTI	Abbreviation for linear time-invariant systems. In LTI systems, if an input consists of the sum of a number of signals, then the output is the sum of the system's responses to each signal considered separately [Lyn89].
MMX™ technology	An enhancement to Intel architecture aimed at better performance in multimedia and communications applications. The technology uses four additional data types, eight 64-bit MMX registers, and 57 additional instructions implementing the SIMD (single instruction, multiple data) technique.
MR	One of the modes, indicating the multi-rate variety of the function.
multi-rate	An operation or signal processing system involving signals with multiple sample rates. Decimation and interpolation are examples of multi-rate operations.
not-in-place	A function that performs its operation not-in-place takes its input from a source array and puts its output in a second, destination array.
polyphase	A computationally efficient method for multi-rate filtering. For example, interpolation or decimation.
CCS	A representation of a complex conjugate-symmetric sequence which is easier to use than the Pack or Perm formats.
Pack	A compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real FFT algorithms ("natural" in the sense that bit-reversed order is natural for radix-2 complex FFTs).
Perm	A format for storing the values for the FFT algorithm. RCPerm format stores the values in the order in which the FFT algorithm uses them. That is, the real and imaginary parts of a given sample need not be adjacent.

saturation	Using saturation arithmetic, when a number exceeds the data-range limit for its data type, it saturates to the upper data-range limit. For example, a signed word greater than 7FFFh saturates to 7FFFh. When a number is less than the lower data-range limit, it saturates to the lower data-range. For example, a signed word less than 8000h saturates to 8000h.
sinusoid	See tone.
Streaming SIMD Extensions	The major enhancement to Intel architecture instruction set. Incorporates a group of general-purpose floating-point instructions operating on packed data, additional packed integer instructions, together with cacheability control and state management instructions. These instructions significantly improve performance of applications using compute-intensive processing of floating-point and integer data.
tone	A sinusoid of a given frequency, phase, and magnitude. Tones are used as test signals and as building blocks for more complex signals.
up-sampling	Up-sampling conceptually increases the signal sampling rate by inserting zero-valued samples between neighboring samples of a signal.
window	A mathematical function by which a signal is multiplied to improve the characteristics of some subsequent analysis. Windows are commonly used in FFT-based spectral analysis.

Index

A

D

E

F

G

H

I

L

M

N

T

U

V

W

X

Z