

GENERIC DIGITAL DESIGN INCORPORATED

DSP Development Support Library for TMS320C62x/67x Reference Manual

2001

DSP Development Support Library for TMS320C62x/67x Reference Manual. GDD300. Release 2.00.01

Copyright © 1999-2005 Generic Digital Design Incorporated, All Rights Reserved.

Information in this document is provided in connection with Generic Digital Design Incorporated products. No license, express or implied, to any intellectual property rights is granted by this document. Except as provided in GDDI's Terms and Conditions of Sale or License Agreement for such products, GDDI assumes no liability whatsoever, and disclaims any express or implied warranty, relating to sale and/or use of GDDI products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right.

Designers must not rely on the absence or characteristics of any features or instructions marked `_reserved_` or `_undefined_`. Generic Digital Design reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. Generic Digital Design Incorporated products may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available upon request.

GDDI products are not intended for use in medical, life saving, or life sustaining applications. GDDI may make changes to specifications and product descriptions at any time, without notice.

Generic Digital Design Incorporated, GDDI and the GDDI logo, are registered trademarks of Generic Digital Design Incorporated.

Third-party brands and names are the property of their respective owners.

Printed in the USA

Contents

Introduction	8
Overview.....	8
Header file	8
Programming considerations	10
References.....	11
Transforms	15
Initialize Complex FFT twiddle factors table.....	15
Complex Forward FFT	15
Complex Inverse FFT	16
Complex Forward FFT (bit-reversed input)	17
Complex Inverse FFT (bit-reversed output)	18
Initialize Real FFT twiddle factors table	18
Real Forward FFT.....	19
Real Inverse FFT	20
Initialize FHT twiddle factors table.....	21
Forward/Inverse FHT	21
Convert FHT to real FFT.....	22
Convert real FFT to FHT	23
Initialize FCT twiddle factors table.....	23
Real Forward FCT	24
Real Inverse FCT	25
DSP functions	31
Linear Auto-Covariation.....	31
Linear Cross-Covariation.....	32
Generic Form Difference Equation (IIR Filter).....	33
Generic IIR Filter (Macro)	34
Difference Equation, two zeroes, two poles (Biquad IIR Filter).....	34
Biquad IIR Filter (Macro).....	35
Linear Convolution (FIR Filter)	35
Generic FIR Filter (Macro).....	36
Decimation with FIR filter.....	36
Conversion to dB (decibels)	37
Histogram Accumulation.....	38

Auto-Spectrum Accumulation	39
Cross-Spectrum Accumulation	40
Coherence Function	41
Transfer Function	41
Exponential Averaging	42
Linear Averaging	43
Hanning (cosine) window	43
Hamming window	44
Blackman window	45
Bartlett window	45
Parzen window	46
Welch window	47
Vector operations (real data)	51
Index of a vector entry with maximum magnitude	51
Index of a vector entry with minimum magnitude	52
Index of the maximum vector entry	52
Index of the minimum vector entry	53
Sum of absolute values of vector entries	54
Sum of vector entries	55
L_2 (Euclidean) norm of a vector	55
Copy vector to a vector	56
Swap two vectors	57
Fill a vector with a constant	57
Dot product	58
Dot product (unit index increments)	59
Scale a vector, add to another vector, store to an output vector	59
Scale a vector, store to an output vector	60
Add a constant to a vector, store to an output vector	61
Add entries of two vectors, store to an output vector	62
Subtract entries of two vectors, store to an output vector	63
Multiply entries of two vectors, store to an output vector	64
Divide entries of two vectors, store to an output vector	65
Construct Givens plane rotations	66
Apply Givens plane rotations, store to output vectors	67
Vector operations (complex data)	71
Index of a vector entry with maximum magnitude	71
Index of a vector entry with minimum magnitude	72
Sum of absolute values of vector entries	72
Sum of vector entries	73
L_2 (Euclidean) norm of a vector	74
Copy vector to a vector	74
Swap two vectors	75

Fill a vector with a constant.....	76
Dot product, conjugating first vector.....	76
Dot product.....	77
Scale a vector, add to another vector, store to an output vector.....	78
Scale a vector conjugate, add to another vector, store to an output vector.....	79
Scale a vector, store to an output vector.....	80
Scale a vector by a real scalar, store to an output vector.....	81
Add a constant to a vector, store to an output vector.....	82
Add entries of two vectors, store to an output vector.....	82
Subtract entries of two vectors, store to an output vector.....	83
Multiply entries of two vectors, store to an output vector.....	84
Divide entries of two vectors, store to an output vector.....	85
Construct Givens plane rotations.....	86
Apply Givens plane rotations, store to output vectors.....	87
Data conversions.....	91
Rectangular to polar coordinate transform.....	91
Polar to rectangular coordinate transform.....	92
Combine two real arrays into a complex array.....	93
Split a complex array into two real arrays.....	94
Convert a Q3.12 array into a real array.....	95
Convert a real array into a Q3.12 array.....	95
Convert a Q15 array into a real array.....	96
Convert a real array into a Q15 array.....	97
Convert an INT8 array into a scaled real array.....	98
Convert a scaled real array into an INT8 array.....	98
Convert an INT16 array into a scaled real array.....	99
Convert a scaled real array into an INT16 array.....	100
Convert an INT32 array into a scaled real array.....	101
Convert a scaled real array into an INT32 array.....	102
Scalar operations.....	107
Square root of sum of squares.....	107
Bit-reversed index (16/32 bits).....	107
Sum of absolute values of real and imaginary parts.....	108
Magnitude of a complex number.....	108
Complex number conjugate.....	109
Square root of a complex number.....	109
Complex sign transfer.....	109
Add two complex numbers.....	110
Add two complex numbers, conjugate first number.....	110
Add two complex numbers, conjugate second number.....	111
Subtract two complex numbers.....	111
Subtract two complex numbers, conjugate first number.....	112

Subtract two complex numbers, conjugate second number.....	112
Multiply two complex numbers.....	113
Multiply two complex numbers, conjugate first number.....	113
Multiply two complex numbers, conjugate second number.....	113
Multiply and accumulate	114
Multiply and accumulate, conjugate second number	114
Multiply and accumulate, conjugate third number	115
Divide two complex numbers.....	115
Divide two complex numbers, conjugate first number.....	116
Divide two complex numbers, conjugate second number.....	116
Complex number raised to an integer power.....	117
Complex number raised to a real power	117
Complex number raised to a complex power	117
Base e logarithm of a complex number	118
Base e exponential function of a complex number.....	118
Sine of a complex number	119
Cosine of a complex number	119
Limits	123
Clip a real array entries.....	123
Threshold a real array entries	124
Miscellaneous functions	129
In-place bit-reverse permutation of a real array (16/32 bits).....	129
In-place bit-reverse permutation of a complex array (16/32 bits).....	129
Bit-reverse copy of a real array (16/32 bits).....	130
Bit-reverse copy of a complex array (16/32 bits).....	131
Scale a real array by a reciprocal power of 2	132
Scale a complex array by a reciprocal power of 2.....	132
Conjugated copy of a complex array	133
Negated copy of a real array.....	134
Negated copy of a complex array	134
Square roots of entries of a real array.....	135
Absolute values of entries of a real array	136
Magnitudes of entries of a complex array	136
Reciprocal entries of a real array.....	137
Reciprocal entries of a complex array	138
Parameters of machine real arithmetic	138
Parameters of machine complex arithmetic.....	139
Data generation.....	143
Generate full period of sine	143
Generate full period of cosine.....	143

Set (restore) the state of the random generator	144
Get (save) the state of the random generator	144
Get the maximum integer random number (macro)	145
Integer Uniform Pseudo-Random Number Generator	145
Floating Point U(0,1) Pseudo-Random Number Generator	145
Floating Point U(a,b) Pseudo-Random Number Generator	146
Fill a vector with pseudo-random U(a,b) numbers	146
Index	149

Introduction

Overview

This document describes usage and calling conventions of the functions available in Generic Digital Design Inc. DSP Development Support Library for TI TMS320C6X Digital Signal Processors family (DSPDSL6X). The library is a collection of over 100 functions and macros and intended to be used in various application areas such as DSP, audio, video, linear algebra, engineering, control, robotics, military and consuming.

The library is supported for use in any development environment using TI ANSI C compiler for TMS320C6x DSP.

Level 1 BLAS (Basic Linear Library Subroutines) standard is fully implemented in the library.

Header file

DSPDSL6x header file <dspdsl6x.h> contains the library functions declarations, macros and defines data types which are consistently used by all library functions and macros. A user must `#include <dspdsl6x.h>` to use the library functions.

The header file introduces `real` and `complex` data types which are defined as

```
/** FLOATING POINT DATA **/  
#if      (defined(_SP))          /* SINGLE PRECISION */  
#define  real      float        /* REAL DATA TYPE */  
#define  complex   fcomplex     /* COMPLEX DATA TYPE */  
#endif  
  
#if      (defined(_DP))          /* DOUBLE PRECISION */  
#define  real      double       /* REAL DATA TYPE */  
#define  complex   dcomplex     /* COMPLEX DATA TYPE */  
#endif
```

Further, the complex data types are defined as

```
/** COMPLEX FLOATING POINT DATA **/
```

```
typedef struct {          /* SINGLE PRECISION */
    float re;
    float im;
} fcomplex;

typedef struct {          /* DOUBLE PRECISION */
    double re;
    double im;
} dcomplex;
```

Identifiers `_SP` and `_DP` are used in precision control macros to specify the single and double floating point precision respectively. If no precision control macros have been defined, single precision constant `_SP` will be defined by default.

A programmer may simply use `real` and `complex` data types and specify required precision using precision control macros `_SP` and `_DP`.

NOTE: Current release of the library supports *only* single precision data types.

The header file also introduces `int16` data type to store Q1.15 or Q3.12 fixed point format data and `int8`, `int32` integer data types that are defined to be

```
#if      (!defined (FIXED_POINT_TYPE))

/** 8-BIT INTEGER DATA TYPE  */
#define int8 char          /* 8-bit int */
#define byte char         /* 8-bit int */ /* add synonym */

/** 16-BIT Q1.15/Q3.12 FIXED POINT DATA TYPE  */
#define int16 short int   /* 16-bit int */

/** 32-BIT INTEGER DATA TYPE  */
#define int32 int         /* 32-bit int */
#endif /* FIXED_POINT_TYPE */
```

Syntactically, the `int16` data type is a synonym of the standard C `short` type and the `int32` data type is a synonym of the standard C `int` data type.

Vectors that are operated on by the library functions have two main attributes, namely number of elements (`length`) and increment. The latter specifies how many memory locations of size equal to a vector element, separate two adjacent elements of the vector. Therefore, the total size N that a vector spans in memory is given by the following equation

$$N = ((n - 1) \cdot |i| + 1) \cdot \text{sizeof}(type),$$

where n is number of elements in the vector, i is the absolute value of the vector increment. It is supposed that vector entries are enumerated starting with 0, $x[0]$, $x[1]$, etc. In the case of negative increment of a vector $i < 0$, it is expected to ensure that the input pointer contains address of the last vector element. Specifically, the address should be

$$x' = x + i \cdot (1 - n),$$

where x is the initial vector location address, i is the negative vector increment, and n is the total number of elements to be processed.

Programming considerations

Programming using the library functions is straightforward and does not require any special techniques. The functions are optimized to utilize C6x pipeline and take advantages of its VLIW architecture.

The only issue that requires special attention is real and complex arrays allocation. Due to C6x load/store instructions require addresses to be aligned on a corresponding data type boundary, it is required that real arrays must be aligned on a WORD (4 bytes) boundary (2 LSBs of the address are zeroes), complex arrays must be aligned on a DOUBLE WORD (8 bytes) boundary (3 LSBs of the address are zeroes). 16 and 32 bit integer (fixed point) arrays are required to be aligned on a HALF WORD (2 bytes boundary, 1 LSB of the address is zero) and WORD boundaries correspondingly.

TI C6x code generation tools automatically align standard C data arrays, e.g. an array of type `double` would be aligned by default on a DOUBLE WORD boundary. However, in the case of an array of type `complex` it is not the case, since the type `complex` is a structure. In that situation an user is supposed to use a `DATA_ALIGN()` pragma with appropriate arguments to align in proper way a complex array. For example

```
#pragma DATA_ALIGN(cdata, 0x8); // Align on a DWORD boundary
complex cdata[1024];
```

would align a 1024-entries `cdata` array on a DOUBLE WORD boundary. The array has been allocated at the compile time.

To align a complex array that needs be allocated at the run time, an user is supposed to use RTS function `memalign()` with the alignment boundary byte argument set to `0x8`

```
complex *cdata;
// Allocate cdata, alnging on a DWORD boundary
cdata = (complex *)memalign(0x8, sizeof(complex)*1024);
```

It must be noted that the library's scalar functions of complex argument do not require an address of their complex arguments to be aligned on a DOUBLE WORD boundary, since these functions never use the LDDW instruction

For more information about C67x load/store instructions and data alignment an user is referred to the corresponding TI code generation tools and processor architecture documentation.

References

- DSP Applications with the TMS320 Family. Theory, Algorithms and Implementations, Vol. 1,2,3, Texas Instruments, 1990.
 - H. V. Sorensen, D. L. Jones, M. T. Heideman, C. S. Burrus: Real-valued fast Fourier transform algorithms, IEEE Trans. ASSP, Vol. ASSP-35, No. 6, pp. 849-864, June 1987.
 - R. N. Bracewell: The Hartley Transform, Oxford University Press, 1986.
 - H. V. Sorensen, D. L. Jones, C. S. Burrus, and M. T. Heideman: On computing the discrete Hartley transform, IEEE Trans. on ASSP, vol. ASSP-33, Oct. 1985.
 - P. R. Uniyal: Transforming real-valued sequences: Fast Fourier versus Fast Hartley transform algorithms, IEEE Trans. on SP, vol. SP-42, Nov. 1994.
 - Barry G. Sherlock and Donald M. Monro: Algorithm 749 ACM Trans. on Mathematical Software, ACM TOMS-21, No.4, 1995
 - V. Kapellini, A. G. Constantinidis, P. Emiliani: Digital Filters and their Applications, Academic Press, London, 1978
 - R. K. Otnes, L. Enochson: Applied Time Series Analysis, Vol. I Basic Techniques, Wiley, New York, 1978
 - A. V. Oppenheim, R. W. Shafer: Digital Signal Processing, Prentice Hall, Englewood Cliffs, NJ, 1975
 - C. Lawson, R. Hanson, D. Kincaid, F. Krogh: Basic Linear Algebra Subprograms for FORTRAN Usage, ACM Transactions on Mathematical Software, No. 5, 1979, pp. 308-323.
 - C. Lawson, R. Hanson, D. Kincaid, F. Krogh: Algorithm 539: Basic Linear Algebra Subprograms for FORTRAN Usage, ACM Transactions on Mathematical Software, No. 5, 1979, pp. 324-325.
 - J. J. Dongarra, C. B. Moler, J. R. Bunch, G. W. Stewart: LINPACK User's Guide, Appendix A., SIAM, Philadelphia, 1979.
 - J. R. Rice: Matrix Computations and Mathematical Software, Appendix A., McGraw Hill Book Company, New York, 1981.
 - J. J. Dongarra, J. J. Du Croz, S. Hammarling, R. J. Hanson, An Extended Set of FORTRAN Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software, Vol. 14, No. 1, 1988, pp. 1-17.
 - J. J. Dongarra, J. J. Du Croz, S. Hammarling, R. J. Hanson: An Extended Set of FORTRAN Basic Linear Algebra Subprograms: Model Implementation and Test
-

Programs, ACM Transactions on Mathematical Software Vol. 14, No. 1, 1988, pp. 18-32.

- J. J. Dongarra, J. J. Du Croz, S. Hammarling, R. J. Hanson: An Extended Set of FORTRAN Basic Linear Algebra Subprograms, Technical Memorandum No. 41 (Revision 3), Argonne National Laboratory, September 1986.
 - J. J. Dongarra, J. J. Du Croz, S. Hammarling, R. J. Hanson: Model Implementation and Test Package for the Extended BLAS, Argonne National Laboratory Report, ANL MCS-TM 81, August 1986.
 - D. S. Dodson, J. G. Lewis, Issues relating to extension of the Basic Linear Algebra Subprograms, ACM SIGNUM Newsletter, Vol. 20, No. 1, 1985, pp. 2-18
 - J. J. Dongarra, J. J. Du Croz, S. Hammarling, R. J. Hanson: A Proposal for an Extended Set of FORTRAN Basic Linear Algebra Subprograms, ACM SIGNUM Newsletter, Vol. 20, No. 1, 1985.
 - J. J. Dongarra: Increasing the Performance of Mathematical Software through High-Level Modularity, in: Proceedings of the Sixth International Symposium on Computing Methods in Engineering and Applied Sciences, (Versailles, France), North-Holland, 1984, pp. 239-248.
 - J. J. Dongarra, J. J. Du Croz, I. S. Duff, S. Hammarling: A Set of Level 3 Basic Linear Algebra Subprograms, Preprint No. 1, Argonne National Laboratory, August 1988.
 - J. J. Dongarra, J. J. Du Croz, I. S. Duff, S. Hammarling: A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs, Argonne National Laboratory Report, ANL MCS-P88-2, August 1988
-

TRANSFORMS

Transforms

cfftinit

Initialize Complex FFT twiddle factors table

Syntax

```
void cfftinit (int m, complx *w)
```

Parameters

- **m** base 2 logarithm of FFT size. The minimum value of m is 5.
- **w** pointer to output table of twiddle factors. The minimum length of array w is $N/4 = 2^{m-2}$.

Description

Initializes a table of twiddle factors (cosines, sines) for computation of Complex FFT. The twiddle factors are stored in the table in bit-reversed order, as interleaved $\cos(\text{bitrev}(i))$, $\sin(\text{bitrev}(i))$. The size of the table is equal to $N/4 = 2^{m-2}$ complex entries. Bit-reversed permutation length is equal to $N/4 = 2^{m-2}$, or $(m-2)$ bits.

Notes

The table computed for FFT of size $N = 2^m$ can be used for all FFTs of size $N(k) = 2^k$, where $k = 5, 6, 7, \dots, m$.

cfft

Complex Forward FFT

Syntax

```
void cfft (int m, const complx *w, complx *x)
```

Parameters

- **m** base 2 logarithm of FFT size. The minimum value of m is 5.
- **w** pointer to input table of twiddle factors, which has been initialized by `cfftinit` function .
- **x** pointer to the input/output complex array with minimum length equal to $N = 2^m$.

Description Computes Forward Fast Fourier Transform of a complex array x using the twiddle factors table prepared by `cfftinit` function. The length of the input data is equal to $N = 2^m$ complex entries.

$$X_l = \sum_{k=0}^{N-1} x_k \cdot \exp(-2\pi ikl / N), \quad l = 0, 1, \dots, N-1$$

The output data are stored in the same array, thus the transform is performed in-place. Since the forward transform is not scaled, the inverse transform has to be scaled by N^{-1} .

The function uses split-radix DIT algorithm to compute Forward FFT.

Notes Input data are stored in normal order. Output data are stored in bit-reversed order.

cffti *Complex Inverse FFT*

Syntax `void cffti (int m, const complx *w, complx *x)`

Parameters

- m base 2 logarithm of FFT size. The minimum value of m is 5.
- w pointer to input table of twiddle factors, which has been initialized by `cfftinit` function .
- x pointer to the input/output complex array with minimum length equal to $N = 2^m$.

Description Computes Inverse Fast Fourier Transform of a complex array x using the twiddle factors table prepared by `cfftinit` function. The length of the input data is equal to $N = 2^m$ complex entries.

$$x_l = \sum_{k=0}^{N-1} X_k \cdot \exp(2\pi ikl / N), \quad l = 0, 1, \dots, N-1$$

The function does not scale the output by reciprocal value of the FFT size. The output data are stored in the same array, thus the transform is performed in-place.

The function uses split-radix DIF algorithm to compute Inverse FFT.

Notes Input data are stored in bit-reversed order. Output data are stored in normal order.

cfftibr *Complex Forward FFT (bit-reversed input)*

Syntax `void cfftibr (int m, const complx *w, complx *x)`

Parameters

- `m` base 2 logarithm of FFT size. The minimum value of `m` is 5.
- `w` pointer to input table of twiddle factors, which has been initialized by `cfftinit` function .
- `x` pointer to the input/output complex array with minimum length equal to $N = 2^m$.

Description Computes Forward Fast Fourier Transform of a complex array `x` using the twiddle factors table prepared by `cfftinit` function. The length of the input data is equal to $N = 2^m$ complex entries.

$$X_l = \sum_{k=0}^{N-1} x_k \cdot \exp(-2\pi i k l / N), \quad l = 0, 1, \dots, N-1$$

The output data are stored in the same array, thus the transform is performed in-place. Since the forward transform is not scaled, the inverse transform has to be scaled by N^{-1} .

The function uses split-radix DIF algorithm to compute Forward FFT.

Notes Input data are stored in bit-reversed order. Output data are stored in normal order.

cfftibr *Complex Inverse FFT (bit-reversed output)*

Syntax `void cfftibr (int m, const complx *w, complx`

*x)

- Parameters**
- `m` base 2 logarithm of FFT size. The minimum value of `m` is 5.
 - `w` pointer to input table of twiddle factors, which has been initialized by `cfftinit` function .
 - `x` pointer to the input/output complex array with minimum length equal to $N = 2^m$.

Description Computes Inverse Fast Fourier Transform of a complex array `x` using the twiddle factors table prepared by `cfftinit` function. The length of the input data is equal to $N = 2^m$ complex entries.

$$x_l = \sum_{k=0}^{N-1} X_k \cdot \exp(2\pi i k l / N), \quad l = 0, 1, \dots, N-1$$

The function does not scale the output by reciprocal value of the FFT size. The output data are stored in the same array, thus the transform is performed in-place.

The function uses split-radix DIT algorithm to compute Inverse FFT.

Notes Input data are stored in normal order. Output data are stored in bit-reversed order.

rfftinit *Initialize Real FFT twiddle factors table*

Syntax `void rfftinit (int m, real *w)`

- Parameters**
- `m` base 2 logarithm of FFT size. The minimum value of `m` is 5.
 - `w` pointer to output table of twiddle factors. The minimum length of array `w` is $N/2+2 = 2^{m-1}+2$.

Description Initializes a table of twiddle factors (cosines, sines) for computation of Real FFT. The twiddle factors are stored in the table in normal order, as interleaved `cos(i)`, `sin(i)`. The size of the table is equal to $N/2+2 = 2^{m-1}+2$ real entries.

Notes The table represents a quarter of period of the cosine and sine functions. The table has to be recomputed for transforms with size other than $N = 2^m$.

rfftf *Real Forward FFT*

Syntax `void rfftf (int m, const real *w, real *x)`

Parameters

- `m` base 2 logarithm of FFT size. The minimum value of `m` is 5.
- `w` pointer to input table of twiddle factors, which has been initialized by `rfftfinit` function .
- `x` pointer to the input/output real array with minimum length equal to $N = 2^m$.

Description Computes Forward Fast Fourier Transform of a real array `x` using the twiddle factors table prepared by `rfftfinit` function. The length of the input data is equal to $N = 2^m$ real entries.

$$X_l = \sum_{k=0}^{N-1} x_k \cdot \exp(-2\pi i k l / N), \quad l = 0, 1, \dots, N-1$$

The output data are stored in the same array, thus the transform is performed in-place. Since the output of the transform is complex, it has to be stored in the real input/output array in a specially arranged order; namely, in the first half of the array the real parts of complex output are stored in normal order, the corresponding complex parts are stored in the upper half of the array in reversed index order

$$Re(X_0), Re(X_1), \dots, Re(X_{N/2}), Im(X_{N/2-1}), \dots, Im(X_1)$$

Zeroth entry X_0 and the central entry $X_{N/2}$ are pure real numbers, therefore it is not required to store their imaginary parts.

Since the forward transform is not scaled, the inverse transform has to be scaled by N^{-1} .

The function uses split-radix DIT algorithm to compute Forward FFT.

Notes Input data are stored in normal order. Output data are stored in a special order; bit-reversed permutation performed internally by the function.

rfffti**Real Inverse FFT****Syntax**

```
void rfffti (int m, const real *w, real *x)
```

Parameters

- **m** base 2 logarithm of FFT size. The minimum value of **m** is 5.
- **w** pointer to input table of twiddle factors, which has been initialized by `rffftinit` function .
- **x** pointer to the input/output real array with minimum length equal to $N = 2^m$.

Description

Computes Inverse Fast Fourier Transform of a complex hermitian sequence of input data using the twiddle factors table prepared by `rffftinit` function. The length of the input data is equal to $N = 2^m$ real entries.

$$x_l = \sum_{k=0}^{N-1} X_k \cdot \exp(2\pi i k l / N), \quad l = 0, 1, \dots, N-1$$

The complex input of the transform is stored in the real input/output array in a specially arranged order; namely, in the first half of the array the real parts of complex input are stored in normal order, the corresponding complex parts are stored in the upper half of the array in reversed index order, i.e. in the order produced on output by the Real Forward FFT function

$$Re(X_0), Re(X_1), \dots, Re(X_{N/2}), Im(X_{N/2-1}), \dots, Im(X_1)$$

Zeroth entry X_0 and the central entry $X_{N/2}$ are pure real numbers, therefore it is not required to store their imaginary parts.

The function does not scale the output by reciprocal value of the FFT size. The real output data are stored in the same array, thus the transform is performed in-place.

The function uses split-radix DIF algorithm to compute Inverse FFT.

Notes

Input data are stored in a special order. Output real data are stored in normal order; bit-reversed permutation is performed internally by the function.

fhtinit *Initialize FHT twiddle factors table*

Syntax `void fhtinit (int m, real *w)`

Parameters

- `m` base 2 logarithm of FHT size. The minimum value of `m` is 5.
- `w` pointer to output table of twiddle factors. The minimum length of array `w` is $N/2+2 = 2^{m-1}+2$.

Description Initializes a table of twiddle factors (cosines, sines) for computation of FHT. The twiddle factors are stored in the table in interleaved (cos(i), sin(i)) format. The size of the table is equal to $N/2+2 = 2^{m-1}+2$ real entries.

Notes The table represents a quarter of period of the cosine and sine functions. The table has to be recomputed for transforms with size other than $N = 2^m$.

fht *Forward/Inverse FHT*

Syntax `void fht (int m, const real *w, real *x)`

Parameters

- `m` base 2 logarithm of FHT size. The minimum value of `m` is 5.
- `w` pointer to input table of twiddle factors, which has been initialized by `fhtinit` function.
- `x` pointer to the input/output complex array with minimum length equal to $N = 2^m$.

Description Computes Forward/Inverse Fast Hartley Transform of a real array `x` using the twiddle factors table prepared by `fhtinit` function. The length of the input data is equal to $N = 2^m$ real entries.

$$X_l = \sum_{k=0}^{N-1} x_k \cdot \text{cas}(2\pi k l / N), \quad l = 0, 1, \dots, N-1$$

where $\text{cas}\alpha \equiv \sin\alpha + \cos\alpha$.

The output data are stored in the same array, thus the transform is performed in-place.

Since the forward transform is not scaled, the inverse transform has to be scaled by N^{-1} . Otherwise Hartley transform is symmetric (self-inverse), i.e. it uses the same formula to perform Forward/Inverse Transform.

The function uses split-radix DIT algorithm to compute both the Forward and Inverse FHT.

Notes Input data are stored in normal order. Output data are also stored in normal order; bit-reverse permutation is performed internally by the function.

fht2fft *Convert FHT to real FFT*

Syntax `void fht2fft (int m, real *x)`

Parameters

- `m` base 2 logarithm of FHT/FFT size. The minimum value of `m` is 5.
- `x` pointer to the input/output real array with minimum length equal to $N = 2^m$.

Description Converts an output of Hartley Transform to Real Fourier Transform. Conversion is performed using the following relations between the two transforms

$$\begin{aligned} \operatorname{Re}(F(i)) &= 0.5 \cdot (H(i) + H(N-i)) = E(i) \\ \operatorname{Im}(F(i)) &= 0.5 \cdot (H(i) - H(N-i)) = O(i) \end{aligned}, \quad i = 0, 1, \dots, N-1$$

where $H(i)$ is Hartley Transform, $E(i)$ is an even part of FHT, $O(i)$ is an odd part and $H(0)=H(N)$. $F(i)$ is Fourier transform, $\operatorname{Re}(x)$ and $\operatorname{Im}(x)$ are real and imaginary parts, commensurately, and $N = 2^m$.

Notes On input, array `x` contains output data from `fht` function. On output, `x` contains Real Discrete Fourier Transform of original data stored in the format described for `rfft` function.

fft2fht *Convert real FFT to FHT*

Syntax `void fft2fht (int m, real *x)`

- Parameters**
- `m` base 2 logarithm of FFT/FHT size. The minimum value of `m` is 5.
 - `x` pointer to the input/output real array with minimum length equal to $N = 2^m$.

Description Converts an output of Real Fourier Transform to Hartley Transform. Conversion is performed using the following relations between the two transforms

$$H(i) = \text{Re}(F(i)) - \text{Im}(F(i)), \quad i = 0, 1, \dots, N-1$$

where $H(i)$ is Hartley Transform, $F(i)$ is Fourier transform, $\text{Re}(x)$ and $\text{Im}(x)$ are real and imaginary parts, commensurately, and $N = 2^m$.

Notes On input, array `x` contains output data from `rfftf` function. On output, `x` contains Discrete Hartley Transform of original data stored in normal order.

fctinit *Initialize FCT twiddle factors table*

Syntax `void fctinit (int m, real *w)`

- Parameters**
- `m` base 2 logarithm of FCT size. The minimum value of `m` is 3.
 - `w` pointer to output table of twiddle factors. The minimum length of array `w` is $N = 2^m$.

Description Initializes a table of twiddle factors for computation of FCT. The size of the table is equal to $N = 2^m$ real entries. Twiddle factors are stored in the table in scaled form, specially index ordered.

Notes The table computed for FCT of size 2^m can be used for all FCTs of size $N(k) = 2^k$, where $k = 3, 4, 5, \dots, m$.

fctf *Real Forward FCT*

Syntax `void fctf (int m, const real *w, real *x, real *work)`

Parameters

- `m` base 2 logarithm of FCT size. The minimum value of `m` is 3.
- `w` pointer to input table of twiddle factors, which has been initialized by `fctinit` function .
- `x` pointer to the input/output real array with minimum length equal to 2^m .
- `work` pointer to a real scratch array with minimum length equal to 2^{m-1} .

Description Computes Forward Fast Cosine Transform of a real array `x` using the twiddle factors table prepared by `fctinit` function. The length of the input data is equal to $N = 2^m$ real entries. The transform is performed using the formula

$$X_l = 2N^{-1} e(l) \sum_{k=0}^{N-1} x_k \cdot \cos\left(\frac{(2k+1)\pi l}{2N}\right), \quad l = 0, 1, \dots, N-1$$

$$\text{where } e(l) = \begin{cases} 2^{-1/2} & \text{if } l = 0 \\ 1 & \text{otherwise} \end{cases}$$

The output data are stored in the same array, thus the transform is performed in-place.

The function uses direct factorization algorithm to compute Forward FCT.

Notes Input data are stored in normal order. Output data are also stored in normal order; even-odd permutation is performed internally by the function.

fcti *Real Inverse FCT***Syntax**

```
void fcti (int m, const real *w, real *x, real
*work)
```

Parameters

- `m` base 2 logarithm of FCT size. The minimum value of `m` is 3.
- `w` pointer to input table of twiddle factors, which has been initialized by `fctinit` function .
- `x` pointer to the input/output real array with minimum length equal to 2^m .
- `work` pointer to a real scratch array with minimum length equal to 2^{m-1} .

Description

Computes Inverse Fast Cosine Transform of a real array `x` using the twiddle factors table prepared by `fctinit` function. The length of the input data is equal to $N = 2^m$ real entries. The transform is performed using the formula

$$x_k = \sum_{l=0}^{N-1} e(l) \cdot X_l \cdot \cos\left(\frac{(2k+1)\pi l}{2N}\right), \quad k = 0, 1, \dots, N-1$$

$$\text{where } e(l) = \begin{cases} 2^{-1/2} & \text{if } l = 0 \\ 1 & \text{otherwise} \end{cases}$$

The output data are stored in the same array, thus the transform is performed in-place.

The function uses direct factorization algorithm to compute Inverse FCT.

Notes

Input data are stored in normal order. Output data are also stored in normal order; even-odd permutation is performed internally by the function.

DSP FUNCTIONS

DSP functions

autocov *Linear Auto-Covariation*

Syntax `void autocov (int n, int m, const real *x,
 int incx, real *z, int incz)`

- Parameters**
- `n` number of lags and number of entries in array `z`. The minimum value of `n` is 4, `n` must be even.
 - `m` number of entries in input array `x`. The minimum value of `m` is 4, `m` must be even and `m > n`.
 - `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
 - `incx` index increment in `x`.
 - `z` pointer to output array `z`. If `incz < 0`, `z` should point to the last entry of array.
 - `incz` index increment in `z`.

Description Calculates linear autocovariation function of a series of data stored in array `x`

$$z_i = \sum_{k=0}^{m-1} x_k \cdot x_{k+i}, \quad i = 0, 1, \dots, n-1$$

It is required for the function to work correctly that the number of entries `m` in the array `x` would be greater than the number of lags `n`.

Notes The actual size of `x` must be larger at least by $(n-1) \cdot \text{incx}$ entries, taking into account spacing between consecutive entries specified by `incx`. The last $(n-1)$ entries in `x` are used for computation of the last $(n-1)$ autocovariations.

crosscov *Linear Cross-Covariation***Syntax**

```
void crosscov (int n, int m, const real *x,
              int incx, const real *y, int incy, real
              *z, int incz)
```

Parameters

- **n** number of entries in array *z*. The minimum value of *n* is 4, *n* must be even.
- **m** number of entries in input arrays *x* and *y*. The minimum value of *m* is 4, *m* must be even and *m* > *n*.
- **x** pointer to input array *x*. If *incx* < 0, *x* should point to the last entry of array. The array is unchanged on return.
- **incx** index increment in *x*.
- **y** pointer to input array *y*. If *incy* < 0, *y* should point to the last entry of array. The array is unchanged on return.
- **incy** index increment in *y*.
- **z** pointer to output array *z*. If *incz* < 0, *z* should point to the last entry of array.
- **incz** index increment in *z*.

Description

Calculates linear crosscovariation function of a series of data stored in arrays *x* and *y*.

$$z_i = \sum_{k=0}^{m-1} x_k \cdot y_{k+i}, \quad i = 0, 1, \dots, n-1$$

It is required for the function to work correctly that the number of entries *m* in the arrays *x* and *y* would be greater than the number of lags *n*.

Notes

The actual size of *x* and *y* must be larger at least by $(n-1) \cdot \text{incx}$ and $(n-1) \cdot \text{incy}$ entries, taking into account spacing between consecutive entries specified by *incx* and *incy*. The last $(n-1)$ entries in *x* and *y* are used for computation of the last $(n-1)$ crosscovariations.

diffeq

Generic Form Difference Equation (IIR Filter)**Syntax**

```
void diffeq (int n, int k, int m, const real
            *a, const real *b, const real *x, int
            incx, real *y, int incy)
```

Parameters

- **n** number of entries in arrays x and y . The minimum value of n is 1.
- **k** number of entries in array a (moving average coefficients). The minimum value of k is 0.
- **m** ($m-1$) is the number of entries in array b (autoregression coefficients). The minimum value of m is 2.
- **a** pointer to input array of moving average coefficients.
- **b** pointer to input array of autoregression coefficients. They are stored starting from b_1 in the zero location of array b .
- **x** pointer to input array x . If $incx < 0$, x should point to the last entry of array. The array is unchanged on return.
- **incx** index increment in x .
- **y** pointer to input/ output array y . If $incy < 0$, y should point to the last entry of array.
- **incy** index increment in y .

Description

Applies linear difference equation with k zeroes and m poles to series of data stored in arrays x and y

$$y_i = a_0x_i + a_1x_{i-1} + \dots + a_{k-1}x_{i-k+1} - b_1y_{i-1} - b_2y_{i-2} - \dots - b_{m-1}y_{i-m+1}$$

or in condensed form

$$y_i = \sum_{j=0}^{k-1} a_j x_{i-j} - \sum_{j=1}^{m-1} b_j y_{i-j}$$

This equation is also called Autoregression/Moving Average (ARMA) model, autoregression with m coefficients, moving averages with k coefficients.

It is implied that the coefficients in arrays a and b are normalized by b_0 before calling the function. The 0-th coefficient is not stored in array b .

Notes

The actual size of x must be larger at least by $(k-1) \cdot incx$ entries. The first k elements in x are used as initial values. Therefore the input pointer x must point to the $x[(k-1) \cdot incx]$ entry.

The actual size of y must be longer at least by $(m-1)\cdot incy$ entries. The first m elements in y are used as initial values. Therefore the input pointer y must point to the $y[(m-1)\cdot incy]$ entry.

iirf **Generic IIR Filter (Macro)**

Syntax `#define iirf(n, k, m, a, b, x, incx, y, incy) \
 diffeq(n, k, m, a, b, x, incx, y, incy)`

Notes See the description of `diffeq` function above for explanation of the parameters to the macro and its functionality.

diffeq22 **Difference Equation, two zeroes, two poles (Biquad IIR Filter)**

Syntax `void diffeq22 (int n, const real *c, const
 real *x, int incx, real *y, int incy)`

- Parameters**
- n number of entries in arrays x and y . The minimum value of n is 1.
 - c pointer to input array with five entries of autoregression /moving average coefficients. The entries of array are normalized by the 0-th autoregression coefficient.
 - x pointer to input array x . If $incx < 0$, x should point to the last entry of array. The array is unchanged on return.
 - $incx$ index increment in x .
 - y pointer to input/ output array y . If $incy < 0$, y should point to the last entry of array.
 - $incy$ index increment in y .

Description Applies linear difference equation with 2 zeroes and 2 poles to series of data stored in arrays x and y (Biquad IIR filter)

$$y_i = c_0x_i + c_1x_{i-1} + c_2x_{i-2} - c_3y_{i-1} - c_4y_{i-2}$$

This equation is also called Autoregression/Moving Average

(ARMA) model, autoregression with 2 coefficients, moving averages with 3 coefficients.

It is implied that the coefficients in array c are normalized by b_0 before calling the function.

Notes The actual size of x and y must be larger at least by $2 \cdot incx$ and $2 \cdot incy$ entries, respectively. The first 2 elements are used as initial values. Therefore the input pointers must point to the $x[2 \cdot incx]$ and $y[2 \cdot incy]$ entries.

iirf2 *Biquad IIR Filter (Macro)*

Syntax

```
#define iirf2(n, c, x, incx, y, incy) \
    diffeq22(n, c, x, incx, y, incy)
```

Notes See the description of `diffeq22` function above for explanation of the parameters to the macro and its functionality.

convoltn *Linear Convolution (FIR Filter)*

Syntax

```
void convoltn (int n, int m, const real *f,
               int incf, const real *x, int incx, real
               *y, int incy)
```

Parameters

- n number of entries in array y to be calculated. The minimum value of n is 4, n must be even.
- m number of entries in array of FIR coefficients f . The minimum value of m is 4, m must be even.
- f pointer to input array of FIR coefficients.
- $incf$ index increment in f .
- x pointer to input array x . If $incx < 0$, x should point to the last entry of array. The array is unchanged on return.
- $incx$ index increment in x .

- `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
- `incy` index increment in `y`.

Description Calculates linear convolution of a series of data stored in array `x` with predefined array of coefficients `f`

$$y_i = \sum_{k=0}^{m-1} f_k \cdot x_{i-k}, \quad i = 0, 1, \dots, n-1$$

Results of convolution are stored into output array `y`.

Notes The actual size of `x` must be larger at least by $(m-1) \cdot incx$ entries. The first m elements in `x` are used as initial values. Therefore the input pointer `x` must point to the `x[(m-1) \cdot incx]` entry.

fir *Generic FIR Filter (Macro)*

Syntax `#define firf(n, m, f, x, incx, y, incy) \`
`convoltn(n, m, f, 1, x, incx, y, incy)`

Notes See the description of `convoltn` function above for explanation of the parameters to the macro and its functionality.

decmfir *Decimation with FIR filter*

Syntax `void decmfir (int n, int p, int m, const real`
`*f, const real *x, int incx, real *y, int`
`incy)`

Parameters

- `n` number of entries in array `y` to be calculated. The value of `n` must be even and ≥ 4 .
- `p` decimation factor.
- `m` number of entries in array of FIR coefficients `f`. The

value of m must be even and ≥ 4 .

- f pointer to input array of FIR coefficients.
- x pointer to input array x . If $incx < 0$, x should point to the last entry of array. The array is unchanged on return.
- $incx$ index increment in x .
- y pointer to output array y . If $incy < 0$, y should point to the last entry of array.
- $incy$ index increment in y .

Description

Performs decimation by factor p combined with (low-pass) FIR filtering. The input data are stored in array x , the input array f contains coefficients of the FIR filter

$$y_i = \sum_{k=0}^{m-1} f_k \cdot x_{pi-k}, \quad i = 0, 1, \dots, n-1$$

Results of decimation are stored into output array y .

It is supposed that the user-supplied coefficients of the filter define a low-pass FIR filter to achieve the anti-aliasing effect (to prevent frequency aliasing).

Notes

The actual size of x must be larger at least by $(m-1) \cdot incx$ entries. The first m elements in x are used as initial values. Therefore the input pointer x must point to the $x[(m-1) \cdot incx]$ entry.

x20db

Conversion to dB (decibels)

Syntax

```
void x20db (int n, int PVF, real a, const real
           *x, int incx, real *y, int incy)
```

Parameters

- n number of entries in arrays x and y . The minimum value of n is 1.
- PVF power/voltage flag
 - $= 0$ forces power definition conversion, $10 \log_{10}(x)$
 - $\neq 0$ forces voltage definition conversion, $20 \log_{10}(x)$
- a 0dB level value.
- x pointer to input array x . If $incx < 0$, x should point to the last entry of array. The array is unchanged on return.

- `incx` index increment in `x`.
- `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
- `incy` index increment in `y`.

Description

Converts normalized magnitudes of entries of `x` to decibels as it has been specified by the input flag `PVF`

$$y_i = \begin{cases} 10 \log_{10}(x_i / a) & \text{(power def.)} \\ 20 \log_{10}(x_i / a) & \text{(voltage def.)} \end{cases}, \quad i = 0, 1, \dots, n-1$$

The upper formula (power definition) would be used if the flag has been set to 0, otherwise the lower formula (voltage definition) is used.

Results are stored into the output array `y`.

hist***Histogram Accumulation*****Syntax**

```
void hist (int n, int kp2, real a, real b,
          const real *x, int incx, int *h)
```

Parameters

- `n` number of entries in array `x`. The minimum value of `n` is 5.
- `kp2` (`k+2`) the number of entries/bins in array `h`.
- `a` lower bound of histogram interval.
- `b` upper bound of histogram interval.
- `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
- `incx` index increment in `x`.
- `h` pointer to input/output histogram.
-

Description

Accumulates (updates) histogram of input data stored in the entries of input array `x`. The `k+2` bins are updated as follows

$$\begin{aligned}
 h_0 &\leftarrow h_0 + 1, & \text{if } x_i < a \\
 h_j &\leftarrow h_j + 1, & j = 1 + \left\lceil \frac{k}{b-a} \cdot (x_i - a) \right\rceil \text{ if } a \leq x_i < b \\
 h_{k+1} &\leftarrow h_{k+1} + 1, & \text{if } x_i \geq b
 \end{aligned}$$

where $\lceil f \rceil$ means integer part of f .

It is supposed that the lower bound a is strictly less than the upper bound b . Both bounds must be valid single precision numbers.

The size of each bin *inside* the interval bounds is $k(b-a)^{-1}$

On return from the function the left- and rightmost bins h_0 and h_{k+1} would contain the number of data points that are to the left/right of the interval boundaries. The rest of the bins would contain the histogram of points which lie inside the specified interval range.

autospec

Auto-Spectrum Accumulation

Syntax

```
void autospec (int n, const complx *x, int
              incx, real *z)
```

Parameters

- n number of entries in arrays x and z . The minimum value of n is 10.
- x pointer to input array x . If $incx < 0$, x should point to the last entry of array. The array is unchanged on return.
- $incx$ index increment in x .
- z pointer to input/output array z .

Description

Accumulates auto-spectrum of a series of data stored in array x into array z

$$z_i \leftarrow z_i + x_i \cdot x_i^*, \quad i = 0, 1, \dots, n-1$$

where x^* means complex conjugate.

Notes

Index increment in the input/output array z is 1.

crosspec *Cross-Spectrum Accumulation*

Syntax `void crosspec (int n, const complx *x, int incx, const complx *y, int incy, complx *z)`

- Parameters**
- `n` number of entries in arrays `x`, `y` and `z`. The minimum value of `n` is 10.
 - `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to input array `y`. If `incy < 0`, `y` should point to the last entry of array. The array is unchanged on return.
 - `incy` index increment in `y`.
 - `z` pointer to input /output array `z`.

Description Accumulates cross-spectrum of a series of data stored in arrays `x` and `y` into array `z`

$$z_i \leftarrow z_i + x_i \cdot y_i^*, \quad i = 0, 1, \dots, n-1$$

where y^* means complex conjugate.

Notes Index increment in the input/output array `z` is 1.

coherfct *Coherence Function*

Syntax `void coherfct (int n, const real *x, const real *y, const complx *z, real *c)`

- Parameters**
- `n` number of entries in arrays `x`, `y`, `z` and `c`. The minimum value of `n` is 1.
 - `x` pointer to input array `x` (auto-spectrum). The array is unchanged on return.

- `y` pointer to input array `y` (auto-spectrum). The array is unchanged on return.
- `z` pointer to input array `z` (cross-spectrum). The array is unchanged on return.
- `c` pointer to output array `c` (coherence function).

Description

Calculates coherence function of two series from their auto- and cross-spectra, stored in arrays `x`, `y` and `z`

$$c_i = \frac{z_i \cdot z_i^*}{x_i \cdot y_i}, \quad i = 0, 1, \dots, n-1$$

where z^* means complex conjugate. x and y are the auto-spectra of the signals, z is the cross-spectrum. Results are stored into the output array `c`.

Notes

Index increment in the input /output arrays is 1.

transfct***Transfer Function*****Syntax**

```
void transfct (int n, const complx *x, const
               real *y, complx *z)
```

Parameters

- `n` number of entries in arrays `x`, `y` and `z`. The minimum value of `n` is 1.
- `x` pointer to input array `x` (cross -spectrum). The array is unchanged on return.
- `y` pointer to input array `y` (auto-spectrum). The array is unchanged on return.
- `z` pointer to output array `z` (transfer function).

Description

Calculates transfer function of two series from their auto- and cross-spectra stored in arrays `x` and `y`

$$\begin{aligned} Re(z_i) &= Re(x_i)/y_i \\ Im(z_i) &= Im(x_i)/y_i \end{aligned}, \quad i = 0, 1, \dots, n-1$$

y is the auto-spectrum, x is the cross-spectrum. Results are stored into the output array `z`.

Notes Index increment in the input /output arrays is 1.

expavg *Exponential Averaging*

Syntax `void expavg (int n, real alpha, const real *x, int incx, real *y, int incy)`

- Parameters**
- `n` number of entries in arrays `x` and `y`. The minimum value of `n` is 1.
 - `alpha` scalar discount factor
 - `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to input/output array `y`. If `incy < 0`, `y` should point to the last entry of array.
 - `incy` index increment in `y`.

Description Updates nearly exponential average of a series of data stored in array `y` to include an additional array `x`

$$y_i \leftarrow (x_i + (\alpha - 1)y_i) / \alpha, \quad i = 0, 1, \dots, n - 1$$

Results are stored into input/output array `y`.

linavg *Linear Averaging*

Syntax `void linavg (int n, real alpha, const real *x, int incx, real *y, int incy)`

- Parameters**
- `n` number of entries in arrays `x` and `y`. The minimum value of `n` is 1.
 - `alpha` scalar discount factor
-

- `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
- `incx` index increment in `x`.
- `y` pointer to input/output array `y`. If `incy < 0`, `y` should point to the last entry of array.
- `incy` index increment in `y`.

Description

Updates linear average of a series of data stored in array `y` to include additional array `x`

$$y_i \leftarrow (x_i + \alpha y_i) / (1 + \alpha), \quad i = 0, 1, \dots, n-1$$

Results are stored into input/output array `y`.

hanning2***Hanning (cosine) window*****Syntax**

```
void hanning2 (int n, const real *x, int incx,
              real *y, int incy)
```

Parameters

- `n` size of the window and number of entries in arrays `x` and `y`. The minimum value of `n` is 1.
- `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
- `incx` index increment in `x`.
- `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
- `incy` index increment in `y`.

Description

Applies Hanning (also called Hann or cosine) window to time domain data stored in array `x`

$$y_i = 0.5 \cdot \left(1 - \cos\left(\frac{2\pi i}{n-1}\right) \right) \cdot x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array `y`.

hamming2 *Hamming window*

Syntax `void hamming2 (int n, const real *x, int incx,
 real *y, int incy)`

- Parameters**
- `n` size of the window and number of entries in arrays `x` and `y`. The minimum value of `n` is 1.
 - `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
 - `incy` index increment in `y`.

Description Applies Hamming window to time domain data stored in array `x`

$$y_i = \left(0.54 - 0.46 \cdot \cos\left(\frac{2\pi i}{n-1}\right) \right) \cdot x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array `y`.

blackmn2 *Blackman window*

Syntax `void blackmn2 (int n, const real *x, int incx,
 real *y, int incy)`

- Parameters**
- `n` size of the window and number of entries in arrays `x` and `y`. The minimum value of `n` is 1.
 - `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
 - `incy` index increment in `y`.

Description Applies Blackman window to time domain data stored in array x

$$y_i = \left(0.42 - 0.5 \cdot \cos\left(\frac{2\pi i}{n-1}\right) + 0.08 \cdot \cos\left(\frac{4\pi i}{n-1}\right) \right) \cdot x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array y .

bartlet2 *Bartlett window*

Syntax `void bartlet2 (int n, const real *x, int incx, real *y, int incy)`

- Parameters**
- n size of the window and number of entries in arrays x and y . The minimum value of n is 1.
 - x pointer to input array x . If $incx < 0$, x should point to the last entry of array. The array is unchanged on return.
 - $incx$ index increment in x .
 - y pointer to output array y . If $incy < 0$, y should point to the last entry of array.
 - $incy$ index increment in y .

Description Applies Bartlett (also called triangular) window to time domain data stored in array x

$$y_i = \left(1 - \frac{2}{(n-1)} \cdot \left| i - \frac{(n-1)}{2} \right| \right) \cdot x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array y .

parzen2 *Parzen window*

Syntax `void parzen2 (int n, const real *x, int incx,`

```
real *y, int incy)
```

- Parameters**
- `n` size of the window and number of entries in arrays `x` and `y`. The minimum value of `n` is 1.
 - `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
 - `incy` index increment in `y`.

Description Applies Parzen window to time domain data stored in array `x`

$$y_i = \left(1 - \frac{2}{(n+1)} \cdot \left| i - \frac{(n-1)}{2} \right| \right) \cdot x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array `y`.

welch2

Welch window

Syntax `void welch2 (int n, const real *x, int incx, real *y, int incy)`

- Parameters**
- `n` size of the window and number of entries in arrays `x` and `y`. The minimum value of `n` is 1.
 - `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
 - `incy` index increment in `y`.

Description Applies Welch window to time domain data stored in array `x`

$$y_i = \left(1 - \left(\frac{2}{(n+1)} \cdot \left(i - \frac{(n-1)}{2} \right) \right)^2 \right) \cdot x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array y .

VECTOR OPERATIONS (REAL DATA)

Vector operations (real data)

isamax

Index of a vector entry with maximum magnitude

Syntax

```
int isamax (int n, const real *x, int incx)
```

Parameters

- `n` number of entries in vector `x`. The minimum value of `n` is 1.
- `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
- `incx` index increment in `x`.

Description

Finds and returns the index of an entry of the input vector with maximum magnitude (absolute value)

$$k = \min i: |x_i| = \max, \quad i = 0, 1, \dots, n-1$$

If there are several entries in the vector with equal maximum magnitude, the function would return the index of the first such an entry.

Notes

In the case of $n < 1$, the function would return -1.

If vector entries are accessed with negative increment `incx`, the returned index would be counted starting from the *last* entry of the vector, so that the address of the entry is given by the expression

$$x + k * incx$$

where `x` points to the *last* entry of the vector.

This is a BLAS Level 1 function.

isamin *Index of a vector entry with minimum magnitude*

Syntax `int isamin (int n, const real *x, int incx)`

- Parameters**
- `n` number of entries in vector `x`. The minimum value of `n` is 1.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.

Description Finds and returns the index of an entry of the input vector with minimum magnitude (absolute value)

$$k = \min i: |x_i| = \min, \quad i = 0, 1, \dots, n-1$$

If there are several entries in the vector with equal minimum magnitude, the function would return the index of the first such an entry.

Notes In the case of $n < 1$, the function would return -1.

If vector entries are accessed with negative increment `incx`, the returned index would be counted starting from the *last* entry of the vector, so that the address of the entry is given by the expression

$$x + k * incx$$

where `x` points to the *last* entry of the vector.

isamax *Index of the maximum vector entry*

Syntax `int isamax (int n, const real *x, int incx)`

- Parameters**
- `n` number of entries in vector `x`. The minimum value of `n` is 1.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.
-

Description Finds and returns the index of an entry of the input vector with maximum value

$$k = \min i: x_i = \max, \quad i = 0, 1, \dots, n-1$$

If there are several entries in the vector with equal maximum value, the function would return the index of the first such an entry.

Notes In the case of $n < 1$, the function would return -1.

If vector entries are accessed with negative increment `incx`, the returned index would be counted starting from the *last* entry of the vector, so that the address of the entry is given by the expression

$$x + k * incx$$

where `x` points to the *last* entry of the vector.

ismin *Index of the minimum vector entry*

Syntax `int ismin (int n, const real *x, int incx)`

- Parameters**
- `n` number of entries in vector `x`. The minimum value of `n` is 1.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.

Description Finds and returns the index of an entry of the input vector with minimum value

$$k = \min i: x_i = \min, \quad i = 0, 1, \dots, n-1$$

If there are several entries in the vector with equal minimum value, the function would return the index of the first such an entry.

Notes In the case of $n < 1$, the function would return -1.

If vector entries are accessed with negative increment `incx`, the

returned index would be counted starting from the *last* entry of the vector, so that the address of the entry is given by the expression

$$x + k * incx$$

where x points to the *last* entry of the vector.

sasum***Sum of absolute values of vector entries*****Syntax**

```
real sasum (int n, const real *x, int incx)
```

Parameters

- n number of entries in vector x . The minimum value of n is 1.
- x pointer to input vector x . If $incx < 0$, x should point to the last entry of vector. The vector is unchanged on return.
- $incx$ index increment in x .

Description

Computes and returns the sum of absolute values of vector entries.

$$s = \sum_{i=0}^{n-1} |x_i|$$

The function returns value of s .

Notes

In the case of $n < 1$, the function would return 0.

This is a BLAS Level 1 function.

svsum***Sum of vector entries*****Syntax**

```
real svsum (int n, const real *x, int incx)
```

Parameters

- n number of entries in vector x . The minimum value of n is 1.

- `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
- `incx` index increment in `x`.

Description Computes and returns the sum of vector entries.

$$s = \sum_{i=0}^{n-1} x_i$$

The function returns value of s .

Notes In the case of $n < 1$, the function would return 0.

snrm2 ***L_2 (Euclidean) norm of a vector***

Syntax `real snrm2 (int n, const real *x, int incx)`

- Parameters**
- `n` number of entries in vector `x`. The minimum value of `n` is 1.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.

Description Computes and returns Euclidean (L_2) norm of input vector.

$$s = \left(\sum_{i=0}^{n-1} |x_i|^2 \right)^{1/2}$$

The function returns value of s .

Notes In the case of $n < 1$, the function would return 0.

This is a BLAS Level 1 function.

scopy *Copy vector to a vector*

Syntax `void scopy (int n, const real *x, int incx,
 real *y, int incy)`

- Parameters**
- `n` number of entries in vectors `x` and `y`. The minimum value of `n` is 1.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output vector `y`. If `incy < 0`, `y` should point to the last entry of vector.
 - `incy` index increment in `y`.

Description Copies entries of the input vector to output vector

$$y_i \leftarrow x_i, \quad i = 0, 1, \dots, n-1$$

Notes In the case of $n < 1$, the function would return immediately.

This is a BLAS Level 1 function.

sswap *Swap two vectors*

Syntax `void sswap (int n, real *x, int incx, real *y,
 int incy)`

- Parameters**
- `n` number of entries in vectors `x` and `y`. The minimum value of `n` is 1.
 - `x` pointer to input/output vector `x`. If `incx < 0`, `x` should point to the last entry of vector.
 - `incx` index increment in `x`.
 - `y` pointer to input/output vector `y`. If `incy < 0`, `y` should point to the last entry of vector.
-

- `incy` index increment in y .

Description Interchanges entries of two vectors

$$x_i \leftrightarrow y_i, \quad i = 0, 1, \dots, n-1$$

Notes In the case of $n < 1$, the function would return immediately.
This is a BLAS Level 1 function.

sfill *Fill a vector with a constant*

Syntax `void sfill (int n, real a, real *x, int incx)`

Parameters

- `n` number of entries in vector x . The minimum value of `n` is 1.
- `a` initializing scalar.
- `x` pointer to output vector x . If `incx < 0`, `x` should point to the last entry of vector.
- `incx` index increment in x .

Description Loads a constant into all entries of the output vector

$$x_i \leftarrow a, \quad i = 0, 1, \dots, n-1$$

Notes In the case of $n < 1$, the function would return immediately.

sdot *Dot product*

Syntax `real sdot (int n, const real *x, int incx,
const real *y, int incy)`

- Parameters**
- n number of entries in vectors x and y . The minimum value of n is 1.
 - x pointer to input vector x . If $incx < 0$, x should point to the last entry of vector. The vector is unchanged on return.
 - $incx$ index increment in x .
 - y pointer to input vector y . If $incy < 0$, y should point to the last entry of vector. The vector is unchanged on return.
 - $incy$ index increment in y .

Description Computes and returns dot product of two vectors

$$s = \sum_{i=0}^{n-1} x_i \cdot y_i$$

The function returns value of s .

Notes In the case of $n < 1$, the function would return immediately.
This is a BLAS Level 1 function.

dotprod *Dot product (unit index increments)*

Syntax `real dotprod (int n, const real *x, const real *y)`

- Parameters**
- n number of entries in arrays x and y . The minimum value of n is 8, and must be even.
 - x pointer to input array x . The array must be aligned on a DWORD boundary (3LSBs are zero). Unchanged on return.
 - y pointer to input array y . The array must be aligned on a DWORD boundary (3LSBs are zero). Unchanged on return.

Description Computes and returns dot product of two vectors

$$s = \sum_{i=0}^{n-1} x_i \cdot y_i$$

The function returns value of s .

Notes The function does not test input parameter n for invalid values.

saxpy2 *Scale a vector, add to another vector, store to an output vector*

Syntax

```
void saxpy2 (int n, real a, const real *x, int
            incx, const real *y, int incy, real *z,
            int incz)
```

```
#define saxpy(n, a, x, incx, y, incy) \
    saxpy2(n, a, x, incx, y, incy, y, incy)
```

- Parameters**
- n number of entries in vectors x and y . The minimum value of n is 1.
 - a scaling scalar.
 - x pointer to input vector x . If $incx < 0$, x should point to the last entry of vector. The vector is unchanged on return.
 - $incx$ index increment in x .
 - y pointer to input vector y . If $incy < 0$, y should point to the last entry of vector. The vector is unchanged on return.
 - $incy$ index increment in y .
 - z pointer to output vector z . If $incz < 0$, z should point to the last entry of vector.
 - $incz$ index increment in z .

Description Scales entries of the first input vector and add to the entries of the second vector

$$z_i \leftarrow a \cdot x_i + y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector z .

The macro `saxpy` (a BLAS Level 1 function) combines the input/output vectors

$$y_i \leftarrow a \cdot x_i + y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the second vector.

Notes In the case of $n < 1$, the function would return immediately.

sscal2 *Scale a vector, store to an output vector*

Syntax `void sscal2 (int n, real a, const real *x, int incx, real *y, int incy)`

```
#define sscal(n,a,x,incx) \
    sscal2(n,a,x,incx,x,incx)
```

Parameters

- `n` number of entries in vector `x`. The minimum value of `n` is 1.
- `a` scaling scalar.
- `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
- `incx` index increment in `x`.
- `y` pointer to output vector `y`. If `incy < 0`, `y` should point to the last entry of vector.
- `incy` index increment in `y`.

Description Scales entries of the input vector by a constant

$$y_i \leftarrow a \cdot x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector `y`.

The macro `sscal` (a BLAS Level 1 function) combines the input/output vectors

$$x_i \leftarrow a \cdot x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the same vector.

Notes In the case of $n < 1$, the function would return immediately.

sshift2 *Add a constant to a vector, store to an output vector*

Syntax

```
void sshift2 (int n, real a, const real *x,  
             int incx, real *y, int incy)
```

```
#define sshift(n,a,x,incx) \  
sshift2(n,a,x,incx,x,incx)
```

- Parameters**
- `n` number of entries in vector `x`. The minimum value of `n` is 1.
 - `a` shifting scalar.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output vector `y`. If `incy < 0`, `y` should point to the last entry of vector.
 - `incy` index increment in `y`.

Description Adds a constant to all entries of the input vector

$$y_i \leftarrow a + x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector `y`.

The macro `sshift` combines the input/output vectors

$$x_i \leftarrow a + x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the same vector.

Notes In the case of $n < 1$, the function would return immediately.

svadd2 *Add entries of two vectors, store to an output vector***Syntax**

```
void svadd2 (int n, const real *x, int incx,
            const real *y, int incy, real *z, int
            incz)
```

```
#define svadd(n, x, incx, y, incy) \
    svadd2(n, x, incx, y, incy, y, incy)
```

Parameters

- **n** number of entries in vectors **x**, **y** and **z**. The minimum value of **n** is 1.
- **x** pointer to input vector **x**. If **incx** < 0, **x** should point to the last entry of vector. The vector is unchanged on return.
- **incx** index increment in **x**.
- **y** pointer to input vector **y**. If **incy** < 0, **y** should point to the last entry of vector. The vector is unchanged on return.
- **incy** index increment in **y**.
- **z** pointer to output vector **z**. If **incz** < 0, **z** should point to the last entry of vector.
- **incz** index increment in **z**.

Description

Adds entries of two vectors

$$z_i \leftarrow x_i + y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector **z**.

The macro `svadd` combines the input/output vectors

$$y_i \leftarrow x_i + y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the second vector.

Notes

In the case of $n < 1$, the function would return immediately.

svsub2 ***Subtract entries of two vectors, store to an output vector*****Syntax**

```
void svsub2 (int n, const real *x, int incx,  
            const real *y, int incy, real *z, int  
            incz)
```

```
#define svsub(n, x, incx, y, incy) \  
    svsub2(n, x, incx, y, incy, y, incy)
```

Parameters

- **n** number of entries in vectors **x**, **y** and **z**. The minimum value of **n** is 1.
- **x** pointer to input vector **x**. If **incx** < 0, **x** should point to the last entry of vector. The vector is unchanged on return.
- **incx** index increment in **x**.
- **y** pointer to input vector **y**. If **incy** < 0, **y** should point to the last entry of vector. The vector is unchanged on return.
- **incy** index increment in **y**.
- **z** pointer to output vector **z**. If **incz** < 0, **z** should point to the last entry of vector.
- **incz** index increment in **z**.

Description

Subtracts entries of two vectors

$$z_i \leftarrow x_i - y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector **z**.

The macro `svsub` combines the input/output vectors

$$y_i \leftarrow x_i - y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the second vector.

Notes

In the case of $n < 1$, the function would return immediately.

svmpy2 *Multiply entries of two vectors, store to an output vector***Syntax**

```
void svmpy2 (int n, const real *x, int incx,
            const real *y, int incy, real *z, int
            incz)
```

```
#define svmpy(n, x, incx, y, incy) \
    svmpy2(n, x, incx, y, incy, y, incy)
```

Parameters

- **n** number of entries in vectors **x**, **y** and **z**. The minimum value of **n** is 1.
- **x** pointer to input vector **x**. If **incx** < 0, **x** should point to the last entry of vector. The vector is unchanged on return.
- **incx** index increment in **x**.
- **y** pointer to input vector **y**. If **incy** < 0, **y** should point to the last entry of vector. The vector is unchanged on return.
- **incy** index increment in **y**.
- **z** pointer to output vector **z**. If **incz** < 0, **z** should point to the last entry of vector.
- **incz** index increment in **z**.

Description

Multiplies entries of two vectors

$$z_i \leftarrow x_i \cdot y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector **z**.

The macro `svmpy` combines the input/output vectors

$$y_i \leftarrow x_i \cdot y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the second vector.

Notes

In the case of $n < 1$, the function would return immediately.

svdiv2 *Divide entries of two vectors, store to an output vector***Syntax**

```
void svdiv2 (int n, const real *x, int incx,  
            const real *y, int incy, real *z, int  
            incz)
```

```
#define svdiv(n, x, incx, y, incy) \  
    svdiv2(n, x, incx, y, incy, y, incy)
```

Parameters

- **n** number of entries in vectors **x**, **y** and **z**. The minimum value of **n** is 1.
- **x** pointer to input vector **x**. If **incx** < 0, **x** should point to the last entry of vector. The vector is unchanged on return.
- **incx** index increment in **x**.
- **y** pointer to input vector **y**. If **incy** < 0, **y** should point to the last entry of vector. The vector is unchanged on return.
- **incy** index increment in **y**.
- **z** pointer to output vector **z**. If **incz** < 0, **z** should point to the last entry of vector.
- **incz** index increment in **z**.

Description

Divides entries of two vectors

$$z_i \leftarrow x_i / y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector **z**.

The macro `svdiv` combines the input/output vectors

$$y_i \leftarrow x_i / y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the second vector.

The function does not test values y_i for zero. Result of division by zero would be a signed QNaN.

Notes

In the case of $n < 1$, the function would return immediately.

srotg *Construct Givens plane rotations*

Syntax `void srotg (real *a, real *b, real *c, real *s)`

- Parameters**
- `a` pointer to input a / output r parameter.
 - `b` pointer to input b / output z parameter.
 - `c` pointer to output cosine c of rotation matrix.
 - `s` pointer to output sine s of rotation matrix.

Description Constructs Givens plane rotation matrix, which satisfy to the equation

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}$$

where s and c are the sine and cosine of rotation matrix respectively. Additional parameters r and ρ will be explained below.

Using the input parameters a and b , the function computes two intermediate values

$$\rho = \begin{cases} a/|a| & \text{if } |a| > |b| \\ b/|b| & \text{if } |a| \leq |b| \end{cases} \quad \text{and} \quad r = \rho \sqrt{a^2 + b^2}$$

Given a , b , ρ and r the elements of rotation matrix are computed as

$$c = \begin{cases} 1 & \text{if } r = 0 \\ a/r & \text{otherwise} \end{cases} \quad \text{and} \quad s = \begin{cases} 0 & \text{if } r = 0 \\ b/r & \text{otherwise} \end{cases}$$

The sign constant $\rho = \pm 1$ does not affect the rotations themselves, but is necessary to be defined when the compact storage using single memory location is requested for the elements of rotation matrix, allowing sine and cosine construction to be stable. The function computes an additional parameter z for that purpose

$$z = \begin{cases} s & \text{if } |a| > |b| \\ 1 & \text{if } c = 0 \\ c^{-1} & \text{if } |a| \leq |b| \end{cases}$$

Sine and cosine of rotation matrix can be restored now as

$$c = \begin{cases} \sqrt{1-z^2} & \text{if } |z| < 1 \\ 0 & \text{if } z = 1 \\ z^{-1} & \text{if } |z| > 1 \end{cases} \quad \text{and} \quad s = \begin{cases} z & \text{if } |z| < 1 \\ 1 & \text{if } z = 1 \\ \sqrt{1-c^2} & \text{if } |z| > 1 \end{cases}$$

Notes On return from the function, the memory locations pointed by a , b , c and s would contain r , z , c and s , commensurately.

This is a BLAS Level 1 function.

srot2 *Apply Givens plane rotations, store to output vectors*

Syntax

```
void srot2 (int n, const real *x, int incx,
           const real *y, int incy, real *w, real
           *z, real c, real s)
```

```
#define srot(n,x,incx,y,incy,c,s) \
    srot2(n,x,incx,y,incy,x,y,c,s)
```

- Parameters**
- n number of entries in vectors x , y , w and z . The minimum value of n is 1.
 - x pointer to input vector x . If $incx < 0$, x should point to the last entry of vector. The vector is unchanged on return.
 - $incx$ index increment in x and w .
 - y pointer to input vector y . If $incy < 0$, y should point to the last entry of vector. The vector is unchanged on return.
 - $incy$ index increment in y and z .
 - w pointer to output vector w .
 - z pointer to output vector z .
 - $incz$ index increment in z .
 - c cosine of rotation matrix.
 - s sine of rotation matrix.

Description Applies Givens plane rotation matrix to input vectors, pointed by x and y

$$\begin{pmatrix} w_i \\ z_i \end{pmatrix} \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix}, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vectors w and z .

The macro `srot` (a BLAS Level 1 function) combines the input/output vectors

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix}, \quad i = 0, 1, \dots, n-1$$

Rotation is performed in-place.

Notes

In the case of $n < 1$, the function would return immediately.

Index increments are the same for (x, w) and (y, z) pairs of input/output vectors.

VECTOR OPERATIONS (COMPLEX DATA)

Vector operations (complex data)

icamax *Index of a vector entry with maximum magnitude*

Syntax `int icamax (int n, const complx *x, int incx)`

- Parameters**
- `n` number of entries in vector `x`. The minimum value of `n` is 1.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.

Description Finds and returns the index of an entry of the input vector with maximum magnitude (absolute value)

$$k = \min i: |x_i| = \max, \quad i = 0, 1, \dots, n-1$$

If there are several entries in the vector with equal maximum magnitude, the function would return the index of the first such an entry.

Notes In the case of $n < 1$, the function would return -1.

If vector entries are accessed with negative increment `incx`, the returned index would be counted starting from the *last* entry of the vector, so that the address of the entry is given by the expression

$$x + k * incx$$

where `x` points to the *last* entry of the vector.

This is a BLAS Level 1 function.

icamin***Index of a vector entry with minimum magnitude***

Syntax

```
int icamin (int n, const complx *x, int incx)
```

Parameters

- `n` number of entries in vector `x`. The minimum value of `n` is 1.
- `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
- `incx` index increment in `x`.

Description

Finds and returns the index of an entry of the input vector with minimum magnitude (absolute value)

$$k = \min i: |x_i| = \min, \quad i = 0, 1, \dots, n-1$$

If there are several entries in the vector with equal minimum magnitude, the function would return the index of the first such an entry.

Notes

In the case of $n < 1$, the function would return -1.

If vector entries are accessed with negative increment `incx`, the returned index would be counted starting from the *last* entry of the vector, so that the address of the entry is given by the expression

$$x + k * incx$$

where `x` points to the *last* entry of the vector.

scasum***Sum of absolute values of vector entries***

Syntax

```
real scasum (int n, const complx *x, int incx)
```

Parameters

- `n` number of entries in vector `x`. The minimum value of `n` is 1.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.
-

Description Computes and returns the sum of absolute values of vector entries.

$$s = \sum_{i=0}^{n-1} |x_i|,$$

where $|x_i| = |Re(x_i)| + |Im(x_i)|$. The function returns value of s .

Notes In the case of $n < 1$, the function would return 0.

This is a BLAS Level 1 function.

cvsum *Sum of vector entries*

Syntax `complex cvsum (int n, const complex *x, int incx)`

- Parameters**
- `n` number of entries in vector `x`. The minimum value of `n` is 1.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.

Description Computes and returns the sum of vector entries.

$$s = \sum_{i=0}^{n-1} x_i,$$

The function returns value of s .

Notes In the case of $n < 1$, the function would return 0.

scnorm2 *L₂ (Euclidean) norm of a vector*

Syntax `real scnorm2 (int n, const complex *x, int incx)`

- Parameters**
- `n` number of entries in vector `x`. The minimum value of `n` is 1.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.

Description Computes and returns Euclidean (L_2) norm of input vector.

$$s = \left(\sum_{i=0}^{n-1} |x_i|^2 \right)^{1/2},$$

where $|x_i|^2 = |Re(x_i)|^2 + |Im(x_i)|^2$. The function returns value of s .

Notes In the case of $n < 1$, the function would return 0.

This is a BLAS Level 1 function.

ccopy *Copy vector to a vector*

Syntax `void ccopy (int n, const complex *x, int incx, complex *y, int incy)`

- Parameters**
- `n` number of entries in vectors `x` and `y`. The minimum value of `n` is 1.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output vector `y`. If `incy < 0`, `y` should point to the last entry of vector.
 - `incy` index increment in `y`.

Description Copies entries of the input vector to output vector

$$y_i \leftarrow x_i, \quad i = 0, 1, \dots, n-1$$

Notes In the case of $n < 1$, the function would return immediately.
This is a BLAS Level 1 function.

cswap *Swap two vectors*

Syntax `void cswap (int n, complex *x, int incx, complex *y, int incy)`

Parameters

- `n` number of entries in vectors `x` and `y`. The minimum value of `n` is 1.
- `x` pointer to input/output vector `x`. If `incx < 0`, `x` should point to the last entry of vector.
- `incx` index increment in `x`.
- `y` pointer to input/output vector `y`. If `incy < 0`, `y` should point to the last entry of vector.
- `incy` index increment in `y`.

Description Interchanges entries of two vectors

$$x_i \leftrightarrow y_i, \quad i = 0, 1, \dots, n-1$$

Notes In the case of $n < 1$, the function would return immediately.
This is a BLAS Level 1 function.

cfill *Fill a vector with a constant*

Syntax	<code>void cfill (int n, complx a, complx *x, int incx)</code>
Parameters	<ul style="list-style-type: none"> • <code>n</code> number of entries in vector <code>x</code>. The minimum value of <code>n</code> is 1. • <code>a</code> initializing scalar. • <code>x</code> pointer to output vector <code>x</code>. If <code>incx < 0</code>, <code>x</code> should point to the last entry of vector. • <code>incx</code> index increment in <code>x</code>.
Description	Loads a constant into all entries of the output vector $x_i \leftarrow a, i = 0, 1, \dots, n-1$
Notes	In the case of $n < 1$, the function would return immediately.

cdotc *Dot product, conjugating first vector*

Syntax	<code>complx cdotc (int n, const complx *x, int incx, const complx *y, int incy)</code>
Parameters	<ul style="list-style-type: none"> • <code>n</code> number of entries in vectors <code>x</code> and <code>y</code>. The minimum value of <code>n</code> is 1. • <code>x</code> pointer to input vector <code>x</code>. If <code>incx < 0</code>, <code>x</code> should point to the last entry of vector. The vector is unchanged on return. • <code>incx</code> index increment in <code>x</code>. • <code>y</code> pointer to input vector <code>y</code>. If <code>incy < 0</code>, <code>y</code> should point to the last entry of vector. The vector is unchanged on return. • <code>incy</code> index increment in <code>y</code>.
Description	Computes and returns dot product of two vectors

$$s = \sum_{i=0}^{n-1} x_i^* \cdot y_i$$

conjugating the entries of input vector pointed by x . The function returns value of s .

Notes In the case of $n < 1$, the function would return immediately.
This is a BLAS Level 1 function.

cdotu *Dot product*

Syntax `complex cdotu (int n, const complex *x, int incx, const complex *y, int incy)`

Parameters

- n number of entries in vectors x and y . The minimum value of n is 1.
- x pointer to input vector x . If $incx < 0$, x should point to the last entry of vector. The vector is unchanged on return.
- $incx$ index increment in x .
- y pointer to input vector y . If $incy < 0$, y should point to the last entry of vector. The vector is unchanged on return.
- $incy$ index increment in y .

Description Computes and returns dot product of two vectors

$$s = \sum_{i=0}^{n-1} x_i \cdot y_i$$

The function returns value of s .

Notes In the case of $n < 1$, the function would return immediately.
This is a BLAS Level 1 function.

caxpy2

Scale a vector, add to another vector, store to an output vector

Syntax

```
void caxpy2 (int n, complex a, const complex *x,
            int incx, const complex *y, int incy,
            complex *z, int incz)
```

```
#define caxpy(n, a, x, incx, y, incy) \
    caxpy2(n, a, x, incx, y, incy, y, incy)
```

Parameters

- `n` number of entries in vectors `x` and `y`. The minimum value of `n` is 1.
- `a` scaling scalar.
- `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
- `incx` index increment in `x`.
- `y` pointer to input vector `y`. If `incy < 0`, `y` should point to the last entry of vector. The vector is unchanged on return.
- `incy` index increment in `y`.
- `z` pointer to output vector `z`. If `incz < 0`, `z` should point to the last entry of vector.
- `incz` index increment in `z`.

Description

Scales entries of the first input vector and add to the entries of the second vector

$$z_i \leftarrow a \cdot x_i + y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector `z`.

The macro `caxpy` (a BLAS Level 1 function) combines the input/output vectors

$$y_i \leftarrow a \cdot x_i + y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the second vector.

Notes

In the case of $n < 1$, the function would return immediately.

caxcpy2 *Scale a vector conjugate, add to another vector, store to an output vector*

Syntax

```
void caxcpy2 (int n, complx a, const complx
             *x, int incx, const complx *y, int incy,
             complx *z, int incz)
```

```
#define caxcpy(n,a,x,incx,y,incy) \
    caxcpy2(n,a,x,incx,y,incy,y,incy)
```

- Parameters**
- `n` number of entries in vectors `x` and `y`. The minimum value of `n` is 1.
 - `a` scaling scalar.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to input vector `y`. If `incy < 0`, `y` should point to the last entry of vector. The vector is unchanged on return.
 - `incy` index increment in `y`.
 - `z` pointer to output vector `z`. If `incz < 0`, `z` should point to the last entry of vector.
 - `incz` index increment in `z`.

Description Scales conjugated entries of the first input vector and add to the entries of the second vector

$$z_i \leftarrow a \cdot x_i^* + y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector `z`.

The macro `caxcpy` combines the input/output vectors

$$y_i \leftarrow a \cdot x_i^* + y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the second vector.

Notes In the case of $n < 1$, the function would return immediately.

cscal2 *Scale a vector, store to an output vector*

Syntax `void cscal2 (int n, complx a, const complx *x,
 int incx, complx *y, int incy)`

```
#define cscal(n,a,x,incx) \  
    cscal2(n,a,x,incx,x,incx)
```

- Parameters**
- `n` number of entries in vector `x`. The minimum value of `n` is 1.
 - `a` scaling scalar.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output vector `y`. If `incy < 0`, `y` should point to the last entry of vector.
 - `incy` index increment in `y`.

Description Scales entries of the input vector by a constant

$$y_i \leftarrow a \cdot x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector `y`.

The macro `cscal` (a BLAS Level 1 function) combines the input/output vectors

$$x_i \leftarrow a \cdot x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the same vector.

Notes In the case of $n < 1$, the function would return immediately.

cscal2 *Scale a vector by a real scalar, store to an output vector*

Syntax

```
void cscal2 (int n, real a, const complex *x,
            int incx, complex *y, int incy)
```

```
#define cscal(n,a,x,incx) \
    cscal2(n,a,x,incx,x,incx)
```

- Parameters**
- `n` number of entries in vector `x`. The minimum value of `n` is 1.
 - `a` scaling scalar.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output vector `y`. If `incy < 0`, `y` should point to the last entry of vector.
 - `incy` index increment in `y`.

Description Scales entries of the input vector by a *real* constant

$$y_i \leftarrow a \cdot x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector `y`.

The macro `cscal` (a BLAS Level 1 function) combines the input/output vectors

$$x_i \leftarrow a \cdot x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the same vector.

Notes In the case of $n < 1$, the function would return immediately.

cshift2 *Add a constant to a vector, store to an output vector*

Syntax

```
void cshift2 (int n, complex a, const complex
             *x, int incx, complex *y, int incy)
```

```
#define cshift(n,a,x,incx) \
    cshift2(n,a,x,incx,x,incx)
```

- Parameters**
- `n` number of entries in vector `x`. The minimum value of `n` is 1.
 - `a` shifting scalar.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output vector `y`. If `incy < 0`, `y` should point to the last entry of vector.
 - `incy` index increment in `y`.

Description Adds a constant to all entries of the input vector

$$y_i \leftarrow a + x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector `y`.

The macro `cshift` combines the input/output vectors

$$x_i \leftarrow a + x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the same vector.

Notes In the case of $n < 1$, the function would return immediately.

cvadd2 *Add entries of two vectors, store to an output vector*

Syntax `void cvadd2 (int n, const complex *x, int incx, const complex *y, int incy, complex *z, int incz)`

```
#define cvadd(n,x,incx,y,incy) \
    cvadd2(n,x,incx,y,incy,y,incy)
```

- Parameters**
- `n` number of entries in vectors `x`, `y` and `z`. The minimum value of `n` is 1.

- `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
- `incx` index increment in `x`.
- `y` pointer to input vector `y`. If `incy < 0`, `y` should point to the last entry of vector. The vector is unchanged on return.
- `incy` index increment in `y`.
- `z` pointer to output vector `z`. If `incz < 0`, `z` should point to the last entry of vector.
- `incz` index increment in `z`.

Description Adds entries of two vectors

$$z_i \leftarrow x_i + y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector `z`.

The macro `cvadd` combines the input/output vectors

$$y_i \leftarrow x_i + y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the second vector.

Notes In the case of $n < 1$, the function would return immediately.

cvsub2 *Subtract entries of two vectors, store to an output vector*

Syntax

```
void cvsub2 (int n, const complex *x, int incx,
            const complex *y, int incy, complex *z, int
            incz)
```

```
#define cvsub(n, x, incx, y, incy) \
    cvsub2(n, x, incx, y, incy, y, incy)
```

Parameters

- `n` number of entries in vectors `x`, `y` and `z`. The minimum value of `n` is 1.
- `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
- `incx` index increment in `x`.

- `y` pointer to input vector `y`. If `incy < 0`, `y` should point to the last entry of vector. The vector is unchanged on return.
- `incy` index increment in `y`.
- `z` pointer to output vector `z`. If `incz < 0`, `z` should point to the last entry of vector.
- `incz` index increment in `z`.

Description

Subtracts entries of two vectors

$$z_i \leftarrow x_i - y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector `z`.

The macro `cvsub` combines the input/output vectors

$$y_i \leftarrow x_i - y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the second vector.

Notes

In the case of $n < 1$, the function would return immediately.

cvmpy2

Multiply entries of two vectors, store to an output vector

Syntax

```
void cvmpy2 (int n, const complex *x, int incx,
             const complex *y, int incy, complex *z, int
             incz)
```

```
#define cvmpy(n, x, incx, y, incy) \
    cvmpy2(n, x, incx, y, incy, y, incy)
```

Parameters

- `n` number of entries in vectors `x`, `y` and `z`. The minimum value of `n` is 1.
- `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
- `incx` index increment in `x`.
- `y` pointer to input vector `y`. If `incy < 0`, `y` should point to the last entry of vector. The vector is unchanged on return.
- `incy` index increment in `y`.

- `z` pointer to output vector `z`. If `incz < 0`, `z` should point to the last entry of vector.
- `incz` index increment in `z`.

Description Multiplies entries of two vectors

$$z_i \leftarrow x_i \cdot y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector `z`.

The macro `cvmpy` combines the input/output vectors

$$y_i \leftarrow x_i \cdot y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the second vector.

Notes In the case of $n < 1$, the function would return immediately.

cvdiv2 *Divide entries of two vectors, store to an output vector*

Syntax

```
void cvdiv2 (int n, const complex *x, int incx,
             const complex *y, int incy, complex *z, int
             incz)
```

```
#define cvdiv(n, x, incx, y, incy) \
    cvdiv2(n, x, incx, y, incy, y, incy)
```

- Parameters**
- `n` number of entries in vectors `x`, `y` and `z`. The minimum value of `n` is 1.
 - `x` pointer to input vector `x`. If `incx < 0`, `x` should point to the last entry of vector. The vector is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to input vector `y`. If `incy < 0`, `y` should point to the last entry of vector. The vector is unchanged on return.
 - `incy` index increment in `y`.
 - `z` pointer to output vector `z`. If `incz < 0`, `z` should point to the last entry of vector.
 - `incz` index increment in `z`.

Description Divides entries of two vectors

$$z_i \leftarrow x_i / y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the output vector z .

The macro `cvdiv` combines the input/output vectors

$$y_i \leftarrow x_i / y_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into the second vector.

The function does not test values y_i for zero. Result of division by zero would be a +QNaN in real and imaginary parts.

Notes In the case of $n < 1$, the function would return immediately.

crotg *Construct Givens plane rotations*

Syntax `void crotg (complex *a, complex *b, real *c, complex *s)`

Parameters

- a pointer to input a / output αr parameter.
- b pointer to input b .
- c pointer to output cosine c of rotation matrix.
- s pointer to output sine s of rotation matrix.

Description Constructs Givens plane rotation matrix, which satisfy to the equation

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \alpha r \\ 0 \end{pmatrix}$$

where s and c are the sine and cosine of rotation matrix respectively. Additional parameters α and r will be explained below.

Using the input parameters a and b , the function computes two intermediate values

$$\alpha = a / |a| \quad \text{and} \quad r = \sqrt{|a|^2 + |b|^2}$$

Given a , b , α and r the elements of rotation matrix are computed as

$$c = \begin{cases} 0 & \text{if } |a|=0 \\ |a|/r & \text{otherwise} \end{cases} \quad \text{and} \quad s = \begin{cases} 1 & \text{if } |a|=0 \\ \alpha b^*/r & \text{otherwise} \end{cases}$$

Notes

On return from the function, the memory locations pointed by c and s would contain c and s , commensurately. The parameter pointed by a has been set to αr if $a \neq 0$, otherwise it has been set to the value of input parameter pointed by b .

This is a BLAS Level 1 function.

csrot2***Apply Givens plane rotations, store to output vectors*****Syntax**

```
void csrot2 (int n, const complex *x, int incx,
            const complex *y, int incy, complex *w,
            complex *z, real c, real s)
```

```
#define csrot(n, x, incx, y, incy, c, s) \
    csrot2(n, x, incx, y, incy, x, y, c, s)
```

Parameters

- n number of entries in vectors x , y , w and z . The minimum value of n is 1.
- x pointer to input vector x . If $incx < 0$, x should point to the last entry of vector. The vector is unchanged on return.
- $incx$ index increment in x and w .
- y pointer to input vector y . If $incy < 0$, y should point to the last entry of vector. The vector is unchanged on return.
- $incy$ index increment in y and z .
- w pointer to output vector w .
- z pointer to output vector z .
- $incz$ index increment in z .
- c cosine of rotation matrix.
- s sine of rotation matrix.

Description

Applies Givens plane rotation matrix to input vectors, pointed by x and y

$$\begin{aligned} \operatorname{Re}\begin{pmatrix} w_i \\ z_i \end{pmatrix} &\leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \times \operatorname{Re}\begin{pmatrix} x_i \\ y_i \end{pmatrix} \\ \operatorname{Im}\begin{pmatrix} w_i \\ z_i \end{pmatrix} &\leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \times \operatorname{Im}\begin{pmatrix} x_i \\ y_i \end{pmatrix}, \quad i = 0, 1, \dots, n-1 \end{aligned}$$

Results are stored into the output vectors w and z .

The macro `csrot` (a BLAS Level 1 function) combines the input/output vectors

$$\begin{aligned} \operatorname{Re}\begin{pmatrix} x_i \\ y_i \end{pmatrix} &\leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \times \operatorname{Re}\begin{pmatrix} x_i \\ y_i \end{pmatrix} \\ \operatorname{Im}\begin{pmatrix} x_i \\ y_i \end{pmatrix} &\leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \times \operatorname{Im}\begin{pmatrix} x_i \\ y_i \end{pmatrix}, \quad i = 0, 1, \dots, n-1 \end{aligned}$$

Rotation is performed in-place.

Notes

In the case of $n < 1$, the function would return immediately.

Index increments are the same for (x, w) and (y, z) pairs of input/output vectors.

DATA CONVERSIONS

Data conversions

dec2plr

Rectangular to polar coordinate transform

Syntax

```
void dec2plr (int n, const real *x, int incx,  
             real *y, int incy)
```

Parameters

- `n` number of 2D entries in `x` and `y`. The minimum value of `n` is 1.
- `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
- `incx` index increment in `x`.
- `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
- `incy` index increment in `y`.

Description

Transforms 2D Cartesian (rectangular) coordinates stored in input array `x` into polar coordinate system

$$\begin{aligned}\rho_i &\leftarrow (x_i^2 + y_i^2)^{1/2} \\ \alpha_i &\leftarrow \operatorname{atan}(y_i / x_i), \quad i = 0, 1, \dots, n-1\end{aligned}$$

Results are stored into output array `y`.

Rectangular coordinates are stored in input array `x` in interleaved format (`x`, `y`). Polar coordinates are stored in output array `y` also in interleaved format (`ρ` , `α`).

Index increment in both arrays is applied to 2D coordinates; one can think of it as `2·incx` or `2·incy` if these increments are to be applied to data of type `real`.

Notes

In the case of `n < 1`, the function would return immediately.

plr2dec***Polar to rectangular coordinate transform***

Syntax

```
void plr2dec (int n, const real *x, int incx,  
             real *y, int incy)
```

Parameters

- `n` number of 2D entries in `x` and `y`. The minimum value of `n` is 1.
- `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
- `incx` index increment in `x`.
- `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
- `incy` index increment in `y`.

Description

Transforms 2D polar coordinates stored in input array `x` into Cartesian (rectangular) coordinate system

$$\begin{aligned}x_i &\leftarrow \rho_i \cos\alpha_i \\y_i &\leftarrow \rho_i \sin\alpha_i\end{aligned}, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array `y`.

Polar coordinates are stored in input array `x` in interleaved format (ρ , α). Rectangular coordinates are stored in output array `y` also in interleaved format (`x`, `y`).

Index increment in both arrays is applied to 2D coordinates; one can think of it as $2 \cdot \text{incx}$ or $2 \cdot \text{incy}$ if these increments are to be applied to data of type `real`.

Notes

In the case of $n < 1$, the function would return immediately.

r2cmplx *Combine two real arrays into a complex array*

Syntax `void r2cmplx (int n, const real *x, int incx,
 const real *y, int incy, complx *z, int
 incz)`

- Parameters**
- `n` number of entries in `x`, `y` and `z`. The minimum value of `n` is 1.
 - `x` pointer to input array `x` of *real* parts. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to input array `y` of *imag* parts. If `incy < 0`, `y` should point to the last entry of array. The array is unchanged on return.
 - `incy` index increment in `y`.
 - `z` pointer to output array `z`. If `incz < 0`, `z` should point to the last entry of array.
 - `incz` index increment in `z`.

Description Combines two real input arrays into a complex array

$$\begin{aligned} \operatorname{Re}(z_i) &\leftarrow x_i \\ \operatorname{Im}(z_i) &\leftarrow y_i \end{aligned}, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array `z`.

Real parts are stored in input array `x`. Imaginary parts are stored in input array `y`.

Notes In the case of $n < 1$, the function would return immediately.

cmplx2r *Split a complex array into two real arrays*

Syntax `void cmplx2r (int n, const complx *z, int
 incz, real *x, int incx, real *y, int`

incy)

- Parameters**
- `n` number of entries in `x`, `y` and `z`. The minimum value of `n` is 1.
 - `z` pointer to input array `y`. If `incz < 0`, `z` should point to the last entry of array. The array is unchanged on return.
 - `incz` index increment in `z`.
 - `x` pointer to output array `x` of *real* parts. If `incx < 0`, `x` should point to the last entry of array.
 - `incx` index increment in `x`.
 - `y` pointer to output array `y` of *imag* parts. If `incy < 0`, `y` should point to the last entry of array.
 - `incy` index increment in `y`.

Description Splits a complex array into two real input arrays

$$\begin{aligned} x_i &\leftarrow \operatorname{Re}(z_i) \\ y_i &\leftarrow \operatorname{Im}(z_i), \quad i = 0, 1, \dots, n-1 \end{aligned}$$

Results are stored into output arrays `x` and `y`.

Real parts are stored in input array `x`. Imaginary parts are stored in input array `y`.

Notes In the case of $n < 1$, the function would return immediately.

Q12_2_IEEE754 Convert a Q3.12 array into a real array

Syntax

```
void Q12_2_IEEE754 (int n, const int16 *q,
                   int incq, real *f, int incf)
```

- Parameters**
- `n` number of entries in `q` and `f`. The minimum value of `n` is 1.
 - `q` pointer to input array `q`. If `incq < 0`, `q` should point to the last entry of array. The array is unchanged on return.
 - `incq` index increment in `q`.

- `f` pointer to output array `f`. If `incf < 0`, `f` should point to the last entry of array.
- `incf` index increment in `f`.

Description

Converts entries of input array `q` stored as a Q3.12 signed fractional numbers into IEEE-754 single precision floating point format

$$f_i \leftarrow 2^{-12}((\text{real})q)_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array `f`.

Notes

In the case of $n < 1$, the function would return immediately.

IEEE754_2_Q12 Convert a real array into a Q3.12 array**Syntax**

```
void IEEE754_2_Q12 (int n, const real *f, int
                    incf, int16 *q, int incq)
```

Parameters

- `n` number of entries in `f` and `q`. The minimum value of `n` is 1.
- `f` pointer to input array `f`. If `incf < 0`, `f` should point to the last entry of array. The array is unchanged on return.
- `incf` index increment in `f`.
- `q` pointer to output array `q`. If `incq < 0`, `q` should point to the last entry of array.
- `incq` index increment in `q`.

Description

Converts entries of input array `f` stored in the IEEE-754 single precision floating point format into Q3.12 signed fractional numbers

$$q_i \leftarrow (\text{int } 16)(2^{12} f_i), \quad i = 0, 1, \dots, n-1$$

Results are stored into output array `f`. The entries in the input array must be less than 8 in magnitude. If an entry value exceeds 8 in magnitude, the result of conversion is undefined.

Notes In the case of $n < 1$, the function would return immediately.

Q15_2_IEEE754 *Convert a Q15 array into a real array*

Syntax

```
void Q15_2_IEEE754 (int n, const int16 *q,
                   int incq, real *f, int incf)
```

Parameters

- **n** number of entries in *q* and *f*. The minimum value of *n* is 1.
- **q** pointer to input array *q*. If *incq* < 0, *q* should point to the last entry of array. The array is unchanged on return.
- **incq** index increment in *q*.
- **f** pointer to output array *f*. If *incf* < 0, *f* should point to the last entry of array.
- **incf** index increment in *f*.

Description

Converts entries of input array *q* stored as a Q15 signed fractional numbers into IEEE-754 single precision floating point format

$$f_i \leftarrow 2^{-15}((\text{real})q)_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array *f*.

Notes In the case of $n < 1$, the function would return immediately.

IEEE754_2_Q15 *Convert a real array into a Q15 array*

Syntax

```
void IEEE754_2_Q15 (int n, const real *f, int
                   incf, int16 *q, int incq)
```

Parameters

- **n** number of entries in *f* and *q*. The minimum value of *n* is 1.
 - **f** pointer to input array *f*. If *incf* < 0, *f* should point
-

to the last entry of array. The array is unchanged on return.

- `incf` index increment in `f`.
- `q` pointer to output array `q`. If `incq < 0`, `q` should point to the last entry of array.
- `incq` index increment in `q`.

Description

Converts entries of input array `f` stored in the IEEE-754 single precision floating point format into Q15 signed fractional numbers

$$q_i \leftarrow (\text{int } 16)(2^{15} f_i), \quad i = 0, 1, \dots, n-1$$

Results are stored into output array `f`. The entries in the input array must be less than 1 in magnitude. If an entry value exceeds 1 in magnitude, the result of conversion is undefined.

Notes

In the case of $n < 1$, the function would return immediately.

INT8_2_IEEE754***Convert an INT8 array into a scaled real array*****Syntax**

```
void INT8_2_IEEE754 (int n, real scale,
                    const int8 *i, int inci, real *f, int
                    incf)
```

Parameters

- `n` number of entries in `i` and `f`. The minimum value of `n` is 1.
- `scale` scaling factor.
- `i` pointer to input array `i`. If `inci < 0`, `i` should point to the last entry of array. The array is unchanged on return.
- `inci` index increment in `i`.
- `f` pointer to output array `f`. If `incf < 0`, `f` should point to the last entry of array.
- `incf` index increment in `f`.

Description

Converts entries of input `int8` integer array `i` into IEEE-754 single precision floating point format scaled numbers

$$f_k \leftarrow \text{scale} \cdot (\text{real})i_k, \quad k = 0, 1, \dots, n-1$$

Scaling factor allows to convert input integers into a desired range of floating point numbers.

Results are stored into output array f .

Notes

In the case of $n < 1$, the function would return immediately.

IEEE754_2_INT8 Convert a scaled real array into an INT8 array

Syntax

```
void IEEE754_2_INT8 (int n, real scale,
                    const real *f, int incf, int8 *i, int
                    inci)
```

Parameters

- n number of entries in f and i . The minimum value of n is 1.
- $scale$ scaling factor.
- f pointer to input array f . If $incf < 0$, f should point to the last entry of array. The array is unchanged on return.
- $incf$ index increment in f .
- i pointer to output array i . If $inci < 0$, i should point to the last entry of array.
- $inci$ index increment in i .

Description

Converts scaled entries of input array f stored in the IEEE-754 single precision floating point format into `int8` signed integer numbers

$$i_k \leftarrow (\text{int } 8)(scale \cdot f_k), \quad k = 0, 1, \dots, n-1$$

Scaling factor allows to convert input floating point numbers into a desired range (bit width) of integer numbers.

Results are stored into output array f . The scaled entries in the input array must be less than 2^7 in magnitude. If a scaled input entry value exceeds 8-bit signed integer range in magnitude, the result of conversion is undefined.

Notes In the case of $n < 1$, the function would return immediately.

INT16_2_IEEE754 *Convert an INT16 array into a scaled real array*

Syntax

```
void INT16_2_IEEE754 (int n, real scale,
                     const int16 *i, int inci, real *f, int
                     incf)
```

Parameters

- `n` number of entries in `i` and `f`. The minimum value of `n` is 1.
- `scale` scaling factor.
- `i` pointer to input array `i`. If `inci < 0`, `i` should point to the last entry of array. The array is unchanged on return.
- `inci` index increment in `i`.
- `f` pointer to output array `f`. If `incf < 0`, `f` should point to the last entry of array.
- `incf` index increment in `f`.

Description

Converts entries of input `int16` integer array `i` into IEEE-754 single precision floating point format scaled numbers

$$f_k \leftarrow scale \cdot (\text{real})i_k, \quad k = 0, 1, \dots, n-1$$

Scaling factor allows to convert input integers into a desired range of floating point numbers.

Results are stored into output array `f`.

Notes In the case of $n < 1$, the function would return immediately.

IEEE754_2_INT16 *Convert a scaled real array into an INT16 array*

Syntax

```
void IEEE754_2_INT16 (int n, real scale,
```

```
const real *f, int incf, int16 *i, int
inci)
```

Parameters

- `n` number of entries in `f` and `i`. The minimum value of `n` is 1.
- `scale` scaling factor.
- `f` pointer to input array `f`. If `incf < 0`, `f` should point to the last entry of array. The array is unchanged on return.
- `incf` index increment in `f`.
- `i` pointer to output array `i`. If `inci < 0`, `i` should point to the last entry of array.
- `inci` index increment in `i`.

Description

Converts scaled entries of input array `f` stored in the IEEE-754 single precision floating point format into `int16` signed integer numbers

$$i_k \leftarrow (\text{int } 16)(\text{scale} \cdot f_k), \quad k = 0, 1, \dots, n-1$$

Scaling factor allows to convert input floating point numbers into a desired range (bit width) of integer numbers.

Results are stored into output array `f`. The scaled entries in the input array must be less than 2^{15} in magnitude. If a scaled input entry value exceeds 16-bit signed integer range in magnitude, the result of conversion is undefined.

Notes

In the case of $n < 1$, the function would return immediately.

INT32_2_IEEE754**Convert an INT32 array into a scaled real array****Syntax**

```
void INT32_2_IEEE754 (int n, real scale,
const int32 *i, int inci, real *f, int
incf)
```

Parameters

- `n` number of entries in `i` and `f`. The minimum value of `n` is 1.
- `scale` scaling factor.

- `i` pointer to input array `i`. If `inci < 0`, `i` should point to the last entry of array. The array is unchanged on return.
- `inci` index increment in `i`.
- `f` pointer to output array `f`. If `incf < 0`, `f` should point to the last entry of array.
- `incf` index increment in `f`.

Description

Converts entries of input `int32` integer array `i` into IEEE-754 single precision floating point format scaled numbers

$$f_k \leftarrow scale \cdot (real)i_k, \quad k = 0, 1, \dots, n-1$$

Scaling factor allows to convert input integers into a desired range of floating point numbers.

Results are stored into output array `f`.

Notes

In the case of $n < 1$, the function would return immediately.

IEEE754_2_INT32***Convert a scaled real array into an INT32 array*****Syntax**

```
void IEEE754_2_INT32 (int n, real scale,
                    const real *f, int incf, int32 *i, int
                    inci)
```

Parameters

- `n` number of entries in `f` and `i`. The minimum value of `n` is 1.
- `scale` scaling factor.
- `f` pointer to input array `f`. If `incf < 0`, `f` should point to the last entry of array. The array is unchanged on return.
- `incf` index increment in `f`.
- `i` pointer to output array `i`. If `inci < 0`, `i` should point to the last entry of array.
- `inci` index increment in `i`.

Description

Converts scaled entries of input array f stored in the IEEE-754 single precision floating point format into `int32` signed integer numbers

$$i_k \leftarrow (\text{int } 32)(\text{scale} \cdot f_k), \quad k = 0, 1, \dots, n-1$$

Scaling factor allows to convert input floating point numbers into a desired range (bit width) of integer numbers.

Results are stored into output array f . The scaled entries in the input array must be less than 2^{31} in magnitude. If a scaled input entry value exceeds 32-bit signed integer range in magnitude, the result of conversion is undefined.

Notes

In the case of $n < 1$, the function would return immediately.

SCALAR OPERATIONS

Scalar operations

pythag *Square root of sum of squares*

Syntax `real pythag (real x, real y)`

- Parameters**
- `x` input real scalar.
 - `y` input real scalar.

Description Computes square root of sum of squares of two numbers without overflow or destructive underflow

$$s = (x^2 + y^2)^{1/2}$$

The function returns value of s .

Notes Benign cancellation is possible if x and y are very close.

lbitrev16/32 *Bit-reversed index (16/32 bits)*

Syntax `int16 lbitrev16 (int16 m, int16 i)`

`int32 lbitrev32 (int32 m, int32 i)`

- Parameters**
- `m` bit width of the input index i . The minimum value of m is 2. The maximum value of m is 16 for the `lbitrev16` function and 31 for the `lbitrev32` function.
 - `i` input index, whose bit-reversal is to be computed.

Description Given the number of bits m the functions would return the bit-reverse

of the input integer i with respect to the specified bit width.

cabs1 *Sum of absolute values of real and imaginary parts*

Syntax `real cabs1 (complex x)`

Parameters • x input complex scalar.

Description Computes sum of absolute values of real and imaginary parts of a complex number

$$s = |Re(x)| + |Im(x)|$$

The function returns value of s .

cabs2 *Magnitude of a complex number*

Syntax `real cabs2 (complex x)`

Parameters • x input complex scalar.

Description Computes magnitude of a complex number without overflow or destructive underflow

$$s = \left(|Re(x)|^2 + |Im(x)|^2 \right)^{1/2}$$

The function returns value of s .

conj *Complex number conjugate*

Syntax `complex conj (complex x)`

Parameters

- `x` input complex scalar.

Description Computes and returns a conjugate of the input complex number

$$s = x^*$$

The function returns value of s .

csroot *Square root of a complex number*

Syntax `complex csroot (complex x)`

Parameters

- `x` input complex scalar.

Description Computes and returns square root of the input complex number

$$s = x^{1/2}$$

The function returns value of s .

csign *Complex sign transfer*

Syntax `complex csign (complex x, complex y)`

Parameters

- `x` input complex scalar.
- `y` input complex scalar.

Description Transfers the complex sign of the second input parameter to the first

input parameter

$$s = x \cdot \frac{y}{|y|}$$

The function returns value of s .

No error check is performed against possible zero magnitude of the second parameter y . In that situation the function returns a +QNaN.

cadd *Add two complex numbers*

Syntax `complex cadd (complex x, complex y)`

Parameters

- `x` input complex scalar.
- `y` input complex scalar.

Description Adds two complex numbers

$$s = x + y$$

The function returns value of s .

ccadd *Add two complex numbers, conjugate first number*

Syntax `complex ccadd (complex x, complex y)`

Parameters

- `x` input complex scalar.
- `y` input complex scalar.

Description Adds two complex numbers, conjugating the first number

$$s = x^* + y$$

The function returns value of s .

caddc *Add two complex numbers, conjugate second number*

Syntax `complex caddc (complex x, complex y)`

- Parameters**
- `x` input complex scalar.
 - `y` input complex scalar.

Description Adds two complex numbers, conjugating the second number

$$s = x + y^*$$

The function returns value of s .

csub *Subtract two complex numbers*

Syntax `complex csub (complex x, complex y)`

- Parameters**
- `x` input complex scalar.
 - `y` input complex scalar.

Description Subtracts two complex numbers

$$s = x - y$$

The function returns value of s .

ccsub *Subtract two complex numbers, conjugate first number*

Syntax `complex ccsub (complex x, complex y)`

- Parameters**
- `x` input complex scalar.
-

- `y` input complex scalar.

Description Subtracts two complex numbers, conjugating the first number

$$s = x^* - y$$

The function returns value of s .

csubc *Subtract two complex numbers, conjugate second number*

Syntax `complex csubc (complex x, complex y)`

- Parameters**
- `x` input complex scalar.
 - `y` input complex scalar.

Description Subtracts two complex numbers, conjugating the second number

$$s = x - y^*$$

The function returns value of s .

cmpr *Multiply two complex numbers*

Syntax `complex cmpr (complex x, complex y)`

- Parameters**
- `x` input complex scalar.
 - `y` input complex scalar.

Description Multiplies two complex numbers

$$s = x \cdot y$$

The function returns value of s .

ccmpy *Multiply two complex numbers, conjugate first number*

Syntax `complex ccmpy (complex x, complex y)`

Parameters

- `x` input complex scalar.
- `y` input complex scalar.

Description Multiplies two complex numbers, conjugating the first number

$$s = x^* \cdot y$$

The function returns value of s .

cmpyc *Multiply two complex numbers, conjugate second number*

Syntax `complex cmpyc (complex x, complex y)`

Parameters

- `x` input complex scalar.
- `y` input complex scalar.

Description Multiplies two complex numbers, conjugating the second number

$$s = x \cdot y^*$$

The function returns value of s .

cmac *Multiply and accumulate*

Syntax `complex cmac (complex x, complex y, complex z)`

- Parameters**
- x input complex scalar.
 - y input complex scalar.
 - z input complex scalar.

Description Multiplies two complex numbers, adds the third number

$$s = x \cdot y + z$$

The function returns value of s .

ccmac ***Multiply and accumulate, conjugate second number***

Syntax `complex ccmac (complex x, complex y, complex z)`

- Parameters**
- x input complex scalar.
 - y input complex scalar.
 - z input complex scalar.

Description Multiplies two complex numbers, conjugating the second number, adds the third number

$$s = x \cdot y^* + z$$

The function returns value of s .

cmacc ***Multiply and accumulate, conjugate third number***

Syntax `complex cmacc (complex x, complex y, complex z)`

- Parameters**
- x input complex scalar.
 - y input complex scalar.
 - z input complex scalar.
-

Description Multiplies two complex numbers, adds conjugated third number

$$s = x \cdot y + z^*$$

The function returns value of s .

cdiv *Divide two complex numbers*

Syntax `complex cdiv (complex x, complex y)`

- Parameters**
- x input complex scalar.
 - y input complex scalar.

Description Divides two complex numbers

$$s = x / y$$

The function returns value of s .

No error check is performed against possible zero magnitude of the second parameter y . In that situation the function returns a +QNaN.

ccdiv *Divide two complex numbers, conjugate first number*

Syntax `complex ccdiv (complex x, complex y)`

- Parameters**
- x input complex scalar.
 - y input complex scalar.

Description Divides two complex numbers, conjugating the first number

$$s = x^* / y$$

The function returns value of s .

No error check is performed against possible zero magnitude of the second parameter y . In that situation the function returns a +QNaN.

cdivc ***Divide two complex numbers, conjugate second number***

Syntax `complex cdivc (complex x, complex y)`

- Parameters**
- `x` input complex scalar.
 - `y` input complex scalar.

Description Divides two complex numbers, conjugating the second number

$$s = x / y^*$$

The function returns value of s .

No error check is performed against possible zero magnitude of the second parameter y . In that situation the function returns a +QNaN.

cpowi ***Complex number raised to an integer power***

Syntax `complex cpowi (complex x, int n)`

- Parameters**
- `x` input complex scalar.
 - `n` input integer power.

Description Computes an integer power function of a complex number

$$s = x^n$$

The function returns value of s .

cpowf *Complex number raised to a real power*

Syntax `complex cpowf (complex x, real f)`

- Parameters**
- `x` input complex scalar.
 - `f` input real power.

Description Computes an integer power function of a complex number

$$s = x^f$$

The function returns value of s .

cpowz *Complex number raised to a complex power*

Syntax `complex cpowz (complex x, complex z)`

- Parameters**
- `x` input complex scalar.
 - `z` input complex power.

Description Computes an integer power function of a complex number

$$s = x^z$$

The function returns value of s .

clog *Base e logarithm of a complex number*

Syntax `complex clog (complex x)`

- Parameters**
- `x` input complex scalar.
-

Description Computes natural (base e) logarithm of a complex number

$$s = \ln x$$

The function returns value of s .

cexp *Base e exponential function of a complex number*

Syntax `complex cexp (complex x)`

Parameters

- `x` input complex scalar.

Description Computes base e exponential function of a complex number

$$s = e^x$$

The function returns value of s .

csin *Sine of a complex number*

Syntax `complex csin (complex x)`

Parameters

- `x` input complex scalar.

Description Computes sine of a complex number

$$s = \sin x$$

The function returns value of s .

ccos *Cosine of a complex number*

Syntax `complex ccos (complex x)`

Parameters

- `x` input complex scalar.

Description Computes cosine of a complex number

$$s = \cos x$$

The function returns value of s .

LIMITS

Limits

vclip2

Clip a real array entries

Syntax

```
void vclip2 (int n, real A, real B, const real
             *x, int incx, real *y, int incy)
```

Parameters

- `n` number of entries in `x` and `y`. The minimum value of `n` is 9.
- `A` lower threshold constant for clipping.
- `B` upper threshold constant for clipping.
- `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
- `incx` index increment in `x`.
- `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
- `incy` index increment in `y`.

Description

Clips entries in the input array to fit their values into the interval defined by the input lower and upper thresholds. It is implicitly assumed that $A < B$. The function does not check this condition.

$$\begin{aligned} \text{if } x_i < A, \text{ then } y_i &= A \\ \text{if } A \leq x_i \leq B, \text{ then } y_i &= x_i, \quad i = 0, 1, \dots, n-1 \\ \text{if } x_i > B, \text{ then } y_i &= B \end{aligned}$$

Results are stored into output array `y`.

In order to keep the smallest entry in `x` unchanged, the lower threshold must be set to negative infinity by calling the `_itof()` macro defined in `<c6x.h>` with hex value `0xff800000` as the parameter.

In order to keep the largest entry in `x` unchanged, the upper threshold must be set to positive infinity by calling the `_itof()` macro defined in `<c6x.h>` with hex value `0x7f800000` as the parameter.

Notes The function does not check the size of input/output arrays to be $n > 9$, it is the application program must set the length is greater than 9 to ensure correct operation of the function.

vthresh2 *Threshold a real array entries*

Syntax `void vthresh2 (int n, real T, real R1, real R2, const real *x, int incx, real *y, int incy)`

Parameters

- `n` number of entries in `x` and `y`. The minimum value of `n` is 9.
- `T` threshold constant.
- `R1` "less than" constant.
- `R2` "greater than or equal to" constant.
- `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
- `incx` index increment in `x`.
- `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
- `incy` index increment in `y`.

Description Binarizes entries in the input array according to the input threshold value T . If the input entry $x_i < T$, then R_1 is stored into output array. If the input entry $x_i \geq T$, then R_2 is stored into output array. The values of R_1 and R_2 input parameters can be arbitrary floating-point numbers.

$$\begin{aligned} &\text{if } x_i < T, \text{ then } y_i = R_1 \\ &\text{if } x_i \geq T, \text{ then } y_i = R_2, \quad i = 0, 1, \dots, n-1 \end{aligned}$$

Results are stored into output array `y`.

Notes The function does not check the size of input/output arrays to be $n > 9$, it is the application program must set the length is greater than 9 to ensure correct operation of the function.

MISCELLANEOUS FUNCTIONS

Miscellaneous functions

sbitrv16/32 *In-place bit-reverse permutation of a real array (16/32 bits)*

Syntax

```
void sbitrv16 (int16 m, real *x)

void sbitrv32 (int32 m, real *x)
```

Parameters

- `m` base 2 logarithm of number of entries in `x`. The minimum value of `m` is 3. The maximum value of `m` is 16 for the `sbitrv16` function and 31 for the `sbitrv32` function.
- `x` pointer to input/output array `x`.

Description

Rearranges entries in the input array in bit-reversed order. The rearranged entries overwrite the input data, the permutation is performed in-place.

$$x_i \leftrightarrow x_{\text{bit-reverse}(i)}, \quad i = 0, 1, \dots, 2^m - 1$$

Notes

In the case of $m < 1$, the function would return immediately.

cbitr16/32 *In-place bit-reverse permutation of a complex array (16/32 bits)*

Syntax

```
void cbitr16 (int16 m, complx *x)

void cbitr32 (int32 m, complx *x)
```

Parameters

- `m` base 2 logarithm of number of entries in `x`. The minimum value of `m` is 3. The maximum value of `m` is 16 for the `cbitr16` function and 31 for the `cbitr32` function.
- `x` pointer to input/output array `x`.

Description Rearranges entries in the input array in bit-reversed order. The rearranged entries overwrite the input data, the permutation is performed in-place.

$$x_i \leftrightarrow x_{\text{bit-reverse}(i)}, \quad i = 0, 1, \dots, 2^m - 1$$

Notes In the case of $m < 1$, the function would return immediately.

sbrcpy16/32 *Bit-reverse copy of a real array (16/32 bits)*

Syntax

```
void sbrcpy16 (int m, const real *x, int incx,
              real *y, int incy)
```

```
void sbrcpy32 (int m, const real *x, int incx,
              real *y, int incy)
```

Parameters

- m base 2 logarithm of number of entries in arrays x and y . The minimum value of m is 3. The maximum value of m is 16 for the `sbrcpy16` function and 31 for the `sbrcpy32` function.
- x pointer to input array x . If $incx < 0$, x should point to the last entry of array. The array is unchanged on return.
- $incx$ index increment in x .
- y pointer to output array y . If $incy < 0$, y should point to the last entry of array.
- $incy$ index increment in y .

Description Copies the entries in the input array in bit-reversed order into the output array.

$$y_i \leftarrow x_{\text{bit-reverse}(i)}, \quad i = 0, 1, \dots, 2^m - 1$$

Results are stored into output array y .

Notes In the case of $m < 1$, the function would return immediately.

cbrcpy16/32 *Bit-reverse copy of a complex array (16/32 bits)*

Syntax

```
void cbrcpy16 (int m, const complx *x, int
              incx, complx *y, int incy)
```

```
void cbrcpy32 (int m, const complx *x, int
              incx, complx *y, int incy)
```

Parameters

- *m* base 2 logarithm of number of entries in arrays *x* and *y*. The minimum value of *m* is 3. The maximum value of *m* is 16 for the `cbrcpy16` function and 31 for the `cbrcpy32` function.
- *x* pointer to input array *x*. If *incx* < 0, *x* should point to the last entry of array. The array is unchanged on return.
- *incx* index increment in *x*.
- *y* pointer to output array *y*. If *incy* < 0, *y* should point to the last entry of array.
- *incy* index increment in *y*.

Description

Copies the entries in the input array in bit-reversed order into the output array.

$$y_i \leftarrow x_{\text{bit-reverse}(i)}, \quad i = 0, 1, \dots, 2^m - 1$$

Results are stored into output array *y*.

Notes

In the case of $m < 1$, the function would return immediately.

sscalrp2 *Scale a real array by a reciprocal power of 2*

Syntax

```
void sscalrp2 (int m, real *x)
```

Parameters	<ul style="list-style-type: none"> • m base 2 logarithm of number of entries in x. The minimum value of m is 3. • x pointer to input/output array x.
Description	<p>Scales entries in the input array by the reciprocal of the array length</p> $x_i = 2^{-m} x_i, \quad i = 0, 1, \dots, 2^m - 1$ <p>The scaling is performed in-place.</p>
Notes	The function does not check the value of $m > 2$, it is the application program must set m to a correct value to ensure correct operation of the function.

cscalrp2 *Scale a complex array by a reciprocal power of 2*

Syntax	<code>void cscalrp2 (int m, complex *x)</code>
Parameters	<ul style="list-style-type: none"> • m base 2 logarithm of number of entries in x. The minimum value of m is 3. • x pointer to input/output array x.
Description	<p>Scales entries in the input array by the reciprocal of the array length</p> $x_i = 2^{-m} x_i, \quad i = 0, 1, \dots, 2^m - 1$ <p>The scaling is performed in-place.</p>
Notes	The function does not check the value of $m > 2$, it is the application program must set m to a correct value to ensure correct operation of the function.

cvconj2 *Conjugated copy of a complex array*

Syntax `void cvconj2 (int n, const complex *x, int
 incx, complex *y, int incy)`

- Parameters**
- `n` number of entries in `x` and `y`. The minimum value of `n` is 1.
 - `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
 - `incy` index increment in `y`.

Description Copies conjugated entries of the input array into the output array.

$$y_i \leftarrow x_i^*, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array `y`.

Notes In the case of $n < 1$, the function would return immediately.

svneg2 *Negated copy of a real array*

Syntax `void svneg2 (int n, const real *x, int incx,
 real *y, int incy)`

- Parameters**
- `n` number of entries in `x` and `y`. The minimum value of `n` is 1.
 - `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
 - `incy` index increment in `y`.
-

Description Copies negated entries of array x to array y

$$y_i \leftarrow -x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array y .

Notes In the case of $n < 1$, the function would return immediately.

cvneg2 *Negated copy of a complex array*

Syntax `void cvneg2 (int n, const complx *x, int incx, complx *y, int incy)`

- Parameters**
- n number of entries in x and y . The minimum value of n is 1.
 - x pointer to input array x . If $incx < 0$, x should point to the last entry of array. The array is unchanged on return.
 - $incx$ index increment in x .
 - y pointer to output array y . If $incy < 0$, y should point to the last entry of array.
 - $incy$ index increment in y .

Description Copies negated entries of array x to array y

$$y_i \leftarrow -x_i, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array y .

Notes In the case of $n < 1$, the function would return immediately.

svsqrt2 *Square roots of entries of a real array*

Syntax `void svsqrt2 (int n, const real *x, int incx,
 real *y, int incy)`

- Parameters**
- `n` number of entries in `x` and `y`. The minimum value of `n` is 1.
 - `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
 - `incy` index increment in `y`.

Description Computes square roots of entries in input array `x`

$$y_i \leftarrow \sqrt{x_i}, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array `y`.

The function does not test negative values x_i . Result of operation with a negative input would be a negative QNaN.

Notes In the case of $n < 1$, the function would return immediately.

svabs2 *Absolute values of entries of a real array*

Syntax `void svabs2 (int n, const real *x, int incx,
 real *y, int incy)`

- Parameters**
- `n` number of entries in `x` and `y`. The minimum value of `n` is 1.
 - `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of vector. The array is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
 - `incy` index increment in `y`.
-

Description Computes absolute values (magnitudes) of the entries in the input array x

$$y_i \leftarrow |x_i|, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array y .

Notes In the case of $n < 1$, the function would return immediately.

cvmagn *Magnitudes of entries of a complex array*

Syntax `void cvmagn (int n, const complx *x, int incx, real *y, int incy)`

- Parameters**
- n number of entries in x and y . The minimum value of n is 1.
 - x pointer to input array x . If $incx < 0$, x should point to the last entry of array. The array is unchanged on return.
 - $incx$ index increment in x .
 - y pointer to output array y . If $incy < 0$, y should point to the last entry of array.
 - $incy$ index increment in y .

Description Computes magnitudes of entries in input array x

$$y_i = \left(\left| \operatorname{Re}(x_i) \right|^2 + \left| \operatorname{Im}(x_i) \right|^2 \right)^{1/2}, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array y .

Notes In the case of $n < 1$, the function would return immediately.

svrcpr2 *Reciprocal entries of a real array*

Syntax `void svrcpr2 (int n, const real *x, int incx,
 real *y, int incy)`

- Parameters**
- `n` number of entries in `x` and `y`. The minimum value of `n` is 1.
 - `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
 - `incx` index increment in `x`.
 - `y` pointer to output array `y`. If `incy < 0`, `y` should point to the last entry of array.
 - `incy` index increment in `y`.

Description Computes reciprocals of entries in input array `x`

$$y_i \leftarrow x_i^{-1}, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array `y`.

The function does not test values x_i for zero. Result of division by zero would be a signed QNaN.

Notes In the case of $n < 1$, the function would return immediately.

cvrcpr2 *Reciprocal entries of a complex array*

Syntax `void cvrcpr2 (int n, const complx *x, int
 incx, complx *y, int incy)`

- Parameters**
- `n` number of entries in `x` and `y`. The minimum value of `n` is 1.
 - `x` pointer to input array `x`. If `incx < 0`, `x` should point to the last entry of array. The array is unchanged on return.
 - `incx` index increment in `x`.
-

- y pointer to output array y . If $incy < 0$, y should point to the last entry of array.
- $incy$ index increment in y .

Description Computes reciprocals of entries in input array x

$$y_i \leftarrow x_i^{-1}, \quad i = 0, 1, \dots, n-1$$

Results are stored into output array y .

The function does not test values x_i for zero. Result of division by zero would be a +QNaN in real and imaginary parts.

Notes In the case of $n < 1$, the function would return immediately.

smach *Parameters of machine real arithmetic*

Syntax `real smach (int job)`

Parameters

- job specifies the parameter to be returned.

Description Compute parameters of the TMS320C6x floating point arithmetic. Machine parameters are set by direct assignment statements.

Consider following machine parameters

- β base of arithmetic
- t number of base b digits
- l the smallest possible exponent
- u the largest possible exponent

then the function would return

- $\varepsilon = \beta^{1-t}$ if $job = 1$
- $x_{tiny} = 10^{+2}\beta^{1+t}$ if $job = 2$
- $x_{huge} = 10^{-2}\beta^{u-t}$ if $job = 3$

Machine epsilon is defined as the minimum positive floating point number such that $1.0 + \varepsilon > 1.0$.

If input parameter `job` is outside its valid set of {1,2,3}, the function would return machine epsilon.

Notes This is a BLAS Level 1 function.

cmach *Parameters of machine complex arithmetic*

Syntax `real cmach (int job)`

Parameters

- `job` specifies the parameter to be returned.

Description Compute parameters of the TMS320C6x floating point arithmetic. Machine parameters are set by direct assignment statements.

Consider following machine parameters

- β base of arithmetic
- t number of base b digits
- l the smallest possible exponent
- u the largest possible exponent

then the function would return

- $\varepsilon = \beta^{1-t}$ if `job` = 1
- $x_{tiny} = 10^{+2}\beta^{1+t}$ if `job` = 2
- $x_{huge} = 10^{-2}\beta^{u-t}$ if `job` = 3

Machine epsilon is defined as the minimum positive floating point number such that $1.0 + \varepsilon > 1.0$.

`cmach` is the same as `smach` except if complex division is done by

$$\frac{1}{x + iy} = \frac{x - iy}{x^2 + y^2}$$

then

$$x_{tiny} = (x_{tiny})^{1/2}$$
$$x_{huge} = (x_{huge})^{1/2}$$

If input parameter `job` is outside its valid set of $\{1,2,3\}$, the function would return machine epsilon.

Notes This is a BLAS Level 1 function.

DATA GENERATION

Data generation

sinwave *Generate full period of sine*

Syntax `void sinwave (int m, real *x)`

- Parameters**
- `m` base 2 logarithm of number of entries in `x`. The minimum value of `m` is 5.
 - `x` pointer to output array.

Description Generates full period of sine function

$$x_k = \sin(2\pi k/2^m), \quad k = 0, 1, \dots, 2^m - 1$$

Notes The function does not check the value of $m > 4$, it is the application program must set m to a correct value to ensure correct operation of the function.

coswave *Generate full period of cosine*

Syntax `void coswave (int m, real *x)`

- Parameters**
- `m` base 2 logarithm of number of entries in `x`. The minimum value of `m` is 5.
 - `x` pointer to output array.

Description Generates full period of cosine function

$$x_k = \cos(2\pi k/2^m), \quad k = 0, 1, \dots, 2^m - 1$$

Notes The function does not check the value of $m > 4$, it is the application program must set m to a correct value to ensure correct operation of the function.

set_seed *Set (restore) the state of the random generator*

Syntax `void set_seed (unsigned int *s)`

Parameters

- `s` pointer to input integer array `s` with 6 entries. The array is unchanged on return.

Description Sets (restores) the state of the uniform PRNG, which has been stored in the input integer array pointed by `s` of length 6. The last entry of `s` must be either 0 or 1.

Effectively this function is used to set the uniform PRNG to a unique state to generate a unique sequence of random numbers or to continue generating of previously interrupted sequence.

get_seed *Get (save) the state of the random generator*

Syntax `void get_seed (unsigned int *s)`

Parameters

- `s` pointer to output integer array `s` with 6 entries.

Description Returns the state of the uniform PRNG, which has been stored in the output integer array pointed by `s` of length 6. The last entry of `s` would be either 0 or 1.

Effectively this function is used to get the state of the uniform PRNG for use it later to continue generating of previously interrupted sequence.

get_rmx *Get the maximum integer random number (macro)*

Syntax `unsigned int get_rmx (void)`

```
#define get_rmx() 4294967286U
```

Description Returns the maximum possible output RND_{max} of the `irandom` integer PRNG.

This function implemented as a macro.

irandom *Integer Uniform Pseudo-Random Number Generator*

Syntax `unsigned int irandom (void)`

Description Returns a pseudo-random integer number uniformly distributed on $(0, RND_{max})$. The maximum random number can be obtained using macro `get_rmx`.

The PRNG implementation is based on lagged Fibonacci numbers.

srandom *Floating Point $U(0,1)$ Pseudo-Random Number Generator*

Syntax `real srandom (void)`

Description Returns a pseudo-random floating point number uniformly distributed on $(0, 1)$ interval. This PRN generator is implemented based on the state of the integer `irandom` PRNG and uses the same algorithm of generating pseudo-random numbers.

Index

A

autocov	29
autospec	37

B

bartlet2	43
blackmn2	42

C

cabs1	104
cabs2	104
cadd	106
cadde	107
caxcpy	77
caxcpy2	77
caxpy	76
caxpy2	76
cbitrv16/32	123
cbrcpy16/32	125
ccadd	106
ccdiv	111
ccmac	110
ccmpy	109
ccopy	72
ccos	115
ccsub	107
cdiv	111
cdive	112
cdotc	74
cdotu	75
cexp	114
cfft	15
cfftbr	17
cfft_i	16
cfftibr	17
cfftinit	15
cfill	74
clog	113
cmac	109
cmacc	110
cmach	133

cmplx2r	91
cmpy	108
cmpyc	109
coherfct	38
conj	105
convoltn	33
coswave	137
cpowf	113
cpowi	112
cpowz	113
crosscov	30
crosspec	38
crotg	84
cscal	78
cscal2	78
cscalrp2	126
cshift	80
cshift2	79
csign	105
csin	114
csroot	105
csrot	85, 86
csrot2	85
csscal	79
csscal2	79
csub	107
csubc	108
cswap	73
cvadd	81
cvadd2	80
cvconj2	127
cvdiv	84
cvdiv2	83
cvmagn	130
cvmpy	83
cvmpy2	82
cvneg2	128
cvrcpr2	131
cvsub	82
cvsub2	81
cvsum	71

D

dec2plr	89
decmfir	34
diffeq	31
diffeq22	32
dotprod	56

E

expavrg	40
---------------	----

F

fctf	24
fcti	25
fctinit	23
fft2fht	23
fht21	
fht2fft	22
fhtinit	21
firf	34

G

get_rmx	139
get_seed	138

H

hamming2	42
hanning2	41
hist	36

I

ibitrev16/32	103
icamax	69
icamin	70
IEEE754_2_INT16	97
IEEE754_2_INT32	99
IEEE754_2_INT8	96
IEEE754_2_Q12	93
IEEE754_2_Q15	94
iirf	32
iirf2	33
INT16_2_IEEE754	97
INT32_2_IEEE754	98
INT8_2_IEEE754	95
irandom	139
isamax	49
isamin	50
ismax	50
ismin	51

L

linavrg	40
---------------	----

P

parzen2	44
plr2dec	90
pythag	103

Q

Q12_2_IEEE754	92
Q15_2_IEEE754	94

R

r2cmplx	91
rfft	19
rfft	20
rfftinit	18
rndvect	140

S

sasum	52
saxpy	57
saxpy2	57
sbitrv16/32	123
sbrcpy16/32	124
scasum	70
scnrm2	72
scopy	54
sdot	55
set_seed	138
sfill	55
sinwave	137
smach	132
snrm2	53
srandom	139
srot	65, 66
srot2	65
srotg	64
sscal	58
sscal2	58
sscalrp2	125
sshift	59
sshift2	59
sswap	54
svabs2	129
svadd	60
svadd2	60
svdiv	63
svdiv2	63
svmpy	62
svmpy2	62
svneg2	127
svrcpr2	131
svsqrt2	129
svsub	61

svsub2.....	61	<i>V</i>	
svsum.....	52	vclip2.....	119
<i>T</i>		vthresh2.....	120
transfct.....	39	<i>W</i>	
<i>U</i>		welch2.....	44
urandom.....	140	<i>X</i>	
		x20db.....	35
