NatureDSP Signal for HiFi5

Digital Signal Processing

Library Reference

Library Release 4.3.1
API Revision 4.31
Updated by IntegrIT November, 2020

IntegrIT, Limited
http://www.integrit.com/
support@integrit.com
Tel: +7 495 545 46 42

IMPORTANT NOTICE

This NatureDSP Signal Processing Library contains copyrighted materials, trade secrets and other proprietary information of IntegrIT, Ltd. This software is licensed for use with Cadence HiFi5 cores only and must not be used for any other processors and platforms. The license to use these sources was given to Cadence, Inc. under Terms and Condition of a Software License Agreement between Cadence, Inc. and IntegrIT, Ltd. Any third party must not use this code in any form unless such rights are given by Cadence, Inc. By compiling, linking, executing or using this library in any form, you unconditionally accept these terms.

Copyright © 2009-2020 IntegrIT, Limited

All Rights Reserved

Table of Contents

IMPORTANT NOTICE 2
Copyright © 2009-2020 IntegrIT, Limited 2
All Rights Reserved 2
Preface 7
About This Manual 7
Supported Targets 7
Notations 7
Abbreviations 7
1 General Library Organization 8
1.1 Headers 8
1.2 Static Variables and Usage of C Standard Libraries 8
1.3 Types 8
1.4 Fractional Formats 9
1.5 Compiler Requirements 9
1.6 Call Conventions 9
1.7 Overflow Control and Intermediate Data Format 9
1.8 Exceptions and Processor Control Registers 10
1.9 Special Numbers 10
1.10 Endianess 10
1.11 Performance Issues 10
1.12 Object Model 11
1.13 Scratch Memory 11
1.14 Brief Function List 12
2 Reference 15
2.1 FIR Filters and Related Functions 15
2.1.1 Block Real FIR Filter 15
2.1.2 Block Real FIR Filter with Arbitrary Parameters 17
2.1.3 Complex Block FIR Filter 19
2.1.4 Decimating Block Real FIR Filter 21
2.1.5 Interpolating Block Real FIR Filter 23
2.1.6 Circular convolution 25
2.1.7 Linear Convolution 26
2.1.8 Circular Correlation 27
2.1.9 Linear Correlation 29
2.1.10 Circular Autocorrelation 29
2.1.11 Linear Autocorrelation 30
2.1.12 Generalized Cross-Correlation Phase Transform (GCC-PHAT) 31
2.1.13 Blockwise Adaptive LMS Algorithm for Real Data 32
2.1.14 2D Convolution 34

2.1.15 Polyphase Analysis FIR Filter 35
2.1.16 Polyphase Synthesis FIR Filter 37

2.2 IIR filters 39
2.2.1 Bi-quad Real Block IIR 39
2.2.2 Lattice Block Real IIR 44
2.2.3 Kalman Filter Update 46

2.3 Mathematics 48
2.3.1 Reciprocal on Q63/Q31/Q15 Numbers 49
2.3.2 Division of Q63/Q31/Q15 Numbers 50
2.3.3 Logarithm 51
2.3.4 Antilogarithm 52
2.3.5 Power Function 53
2.3.6 Square Root 54
2.3.7 Reciprocal Square Root 55
2.3.8 Sine/Cosine 56
2.3.9 Tangent 57
2.3.10 Arctangent 58
2.3.11 Full Quadrant Arctangent 59
2.3.12 Hyperbolic Tangent 60
2.3.13 Sigmoid 60
2.3.14 Rectifier function 61
2.3.15 Softmax 62
2.3.16 Integer to Float Conversion 62
2.3.17 Float to Integer Conversion 63

2.4 Complex Mathematics 64
2.4.1 Complex Magnitude 64

2.5 Vector Operations 65
2.5.1 Vector Dot Product 65
2.5.2 Batch Computation of Vector Dot Products 66
2.5.3 Vector Sum 67
2.5.4 Power of a Vector 67
2.5.5 Vector Scaling with Saturation 68
2.5.6 Common Exponent 70
2.5.7 Vector Min/Max 71

2.6 Emulated Floating Point Operations 72
2.6.1 Vector Addition for Emulated Floating Point 72
2.6.2 Vector Multiply for Emulated Floating Point 73
2.6.3 Vector Multiply-Accumulate for Emulated Floating Point 73
2.6.4 Vector Dot Product for Emulated Floating Point 74

2.7 Matrix Operations 75
2.7.1 Matrix Multiply 75
2.7.2 Matrix by Vector Multiply 78
2.7.3 Vector by Vector Multiply 79
2.7.4 Three Matrices Product 80
2.7.5 Matrix Transpose 81

2.8 Matrix Decomposition/Inversion 83
2.8.1 Gauss-Jordan Matrix Inverse 83
2.8.2 Gauss-Jordan Matrix Equation Solver 84
2.8.3 Cholesky MMSE Solver 86

Copyright 2009-2020, IntegrIT Ltd., www.integrit.com

2.8.4 Cholesky Decomposition for Pseudo-inversion 88
2.8.5 Cholesky Forward Substitution for Pseudo-inversion 90
2.8.6 Cholesky Backward Substitution for Pseudo-inversion 92
2.8.7 Cholesky Presudoinversion 93
2.8.8 Cholesky Preprocessing 95

2.9 Fitting/Interpolation 97
2.9.1 Polynomial Approximation 97

2.10 Fast Fourier Transforms 98
2.10.1 FFT on Complex Data 100
2.10.2 FFT on Real Data 101
2.10.3 Inverse FFT on Complex Data 102
2.10.4 Inverse FFT Forming Real Data 104
2.10.5 FFT on Complex Data with Optimized Memory Usage 105
2.10.6 FFT on Real Data with Optimized Memory Usage 106
2.10.7 Inverse FFT on Complex Data with Optimized Memory Usage 107
2.10.8 Inverse FFT on Real Data with Optimized Memory Usage 109
2.10.9 Power Spectrum 110
2.10.10 Discrete Cosine Transform 111
2.10.11 Modified Discrete Cosine Transform 112
2.10.12 2D Discrete Cosine Transform 113
2.10.13 2D Inverse Discrete Cosine Transform 114

2.11 Mel-Frequency Cepstral Coefficients 114
2.11.1 Compute Log Mel Filterbank Energies 115
2.11.2 Compute Mel-Frequency Cepstrum Coefficients 117
2.11.3 HTK Delta Coefficients 120

2.12 Image Processing 122
2.12.1 Image Padding/Cropping 122
2.12.2 Image Histogram 124
2.12.3 Image Normalization 124
2.12.4 RGB/YUV Conversion 125
2.12.5 Image Interleave/Deinterleave 128
2.12.6 Image Resize 129
2.12.7 Image Rotation 130
2.12.8 Image 2D FFT 132

2.13 Identification Routines 133
2.13.1 Library Version Request 133
2.13.2 Library API Version Request 134
2.13.3 Library API Capability Request 134

3 Test Environment and Examples 135

3.1 Supported Use Environment, Configurations and Targets 135

3.2 Building the NatureDSP Signal Library and the Testdriver 135
3.2.1 Importing the workspaces in Xtensa Xplorer 135
3.2.2 Building and Running Tests 135
3.2.3 Command-line Options 136

4 Appendix 138

4.1 Matlab Code for Conversion of SOS Matrix to Coefficients of IIR Functions 138
4.1.1 bqriir16x16_df1, bqriir32x16_df1 conversion 138

4.1.2 bqriir16x16_df2, bqriir32x16_df2 conversion 139
4.1.3 bqriir32x32_df1 conversion 140
4.1.4 bqriir32x32_df2 conversion 140
4.1.5 bqriirf_df1, bqriirf_df2, bqriirf_df2t conversion 141

4.2 Matlab Code for Generation the Twiddle Tables 142
4.2.1 Twiddles for fft_cplx32x16_ie, ifft_cplx32x16_ie, fft_real32x16_ie, ifft_real32x16_ie, fft_cplx16x16_ie, ifft_cplx16x16_ie, fft_real16x16_ie, ifft_real16x16_ie 142
4.2.2 Twiddles for fft_cplx32x32_ie, ifft_cplx32x32_ie, fft_real32x32_ie, ifft_real32x32_ie 143
4.2.3 Twiddles for fft_cplxf_ie, ifft_cplxf_ie, fft_realf_ie, ifft_realf_ie 143

4.3 Matlab Code for Nonlinear Image Normalization 143

4.4 About IIR filter validation 143

5 Customer Support 145

Preface

About This Manual

Welcome to the **NatureDSP Signal Processing Library**, or **NatureDSP Signal** or library for short. The library is a collection of number highly optimized DSP functions for the DSP targets.

This source code library includes C-callable functions (ANSI-C language compatible) for general signal processing (filtering, correlation, convolution), math and vector functions. Library supports both fixed-point and single precision floating data types.

Supported Targets

Library supports Cadence HiFi5 with VFPU/SFPU little endian targets. Call **IntegrIT** to support more targets or other CPUs.
In general, library API is the same for all supported cores, but some functionality might be missing depending on core abilities. Common rules are:

- functions with floating point inputs/outputs require VFPU/SFPU option of the core

Presence of specific functions might be detected in runtime using library identification routines, see para 2.13.3.

Notations

This document uses the following conventions:

- program listings, program examples, interactive displays, filenames, variables and another software elements are shown in a special typeface (Courier);
- tables use smaller fonts.

Abbreviations

API	Application program interface
DCT	Discrete Cosine Transform
DSP	Digital signal processing
FFT	Fast Fourier transform
FIR	Finite impulse response
IDE	Integrated development environment
IFFT	Inverse Fast Fourier transform
IIR	Infinite impulse response
IR	Impulse response
LMS	Least mean squares
MFCC	Mel frequency cepstrum coefficients

1 General Library Organization

1.1 Headers

The library is delivered in several packages. The API for each package is defined in the appropriate header file which describes particular functions in the package. When the appropriate `#include` preprocessor directive is included in your source, the compiler uses the prototypes to check that each function is called with the correct arguments.

`./library/include/NatureDSP_types.h`	Declarations of basic data types and compiler auto detection	1.3
`./library/include/NatureDSP_Signal.h`	Declarations of all library functions	
`./library/include/NatureDSP_Signal_fir.h`	FIR Filters and Related Functions	2.1
`./library/include/NatureDSP_Signal_iir.h`	IIR Filters	2.2
`./library/include/NatureDSP_Signal_math.h`	Math Functions	2.3
`./library/include/NatureDSP_Signal_complex.h`	Complex Math Functions	2.4
`./library/include/NatureDSP_Signal_vector.h`	Vector Operations	2.5, 2.6
`./library/include/NatureDSP_Signal_matop.h`	Matrix Operations	2.7
`./library/include/NatureDSP_Signal_matinv.h`	Matrix Decomposition and Inversion Functions	2.8
`./library/include/NatureDSP_Signal_fit.h`	Fitting/interpolation	2.9
`./library/include/NatureDSP_Signal_fft.h`	FFT/DCT Routines	2.10
`./library/include/NatureDSP_ Signal_audio.h`	Mel frequency cepstrum coefficients functions	2.11
`./library/include/NatureDSP_ Signal_img.h`	Image processing functions	2.12
`./library/include/NatureDSP_ Signal_id.h`	Identification functions	2.12

1.2 Static Variables and Usage of C Standard Libraries

All library functions are re-entrant. Library functions do not call functions from standard C-library.

1.3 Types

Library uses the following C types with defined length

Name	Description	Alignment, bytes
`int8_t`	8-bit signed value	1
`uint8_t`	8-bit unsigned value	1
`int16_t`	16-bit signed value	2
`f24`	24-bit fractional type	4
`int32_t`	32-bit signed value	4
`uint32_t`	32-bit unsigned value	4
`int64_t`	64-bit signed value	8
`float32_t`	32-bit single precision floating point value	4
`complex_float`	complex single precision floating point (pair of two 32-bit values)	8
`complex_fract16`	complex 16-bit factional value (pair of two 16-bit values)	4
`complex_fract32`	complex 32-bit factional value (pair of two 32-bit values)	8

It is assumed throughout this Reference that constant pointers passed through function arguments point at read-only data

Data of given type should be aligned on its `sizeof()`, see table above.

1.4 Fractional Formats

Natively, HiFi CPU manipulates with standard 16 and 32-bit data. However, for backward compatibility with HiFi2/HiFi3 it supports special fractional type `f24` which is stored in a memory as 32 bit word keeping significant bits in bits 8 through 31. So, from that perspective it may be treated as `Q31` number. But users should take into account that 8 LSB are ignored. Unless specifically noted, library functions use that Q31 format, or, in another words, Q0.31.
In a `Qm.n` format, there are m bits used to represent the two's complement integer portion of the number, and n bits used to represent the two's complement fractional portion. `m+n+1` bits are needed to store a general `Qm.n` number. The extra bit is needed to store the sign of the number in the most-significant bit position. The representable integer range is specified by $[-2^m, 2^m-1]$ and the finest fractional resolution is 2^{-n}. Normally, m from `Q` notation is omitted (because total length is defined of data type used for operand) and it is simply written as `Qn`.

Example data type and their formats are collected in the table below:

Data type	Format	Range	Resolution	Minimum value	Maximum value
int16_t	Q0.15	-1 … 0,999969	3e-5	-32768	32767
int16_t	Q6.9	-64 … 63,998	2e-3	-32768	32767
int32_t	Q1.30	-2 … 1,9999999991	9e-10	-2147483648	2147483647
int32_t	Q0.31	-1 … 0,9999999995	5e-10	-2147483648	2147483647
int32_t	Q6.25	-64… 63,999999970	3e-8	-2147483648	2147483647
int32_t	Q16.15	-65536… 65535,99997	3e-5	-2147483648	2147483647
int32_t	Q23.8	-8388608... 8388607.996	4e-3	-2147483648	2147483647
f24	Q1.30	-2 … 1,9999997625	2e-7	-2147483648	2147483392
f24	Q0.31	-1 … 0,9999998784	1e-7	-2147483648	2147483392
f24	Q6.25	-64… 63,99999240	8e-6	-2147483648	2147483392
f24	Q16.15	-65536…65535,9921875	8e-3	-2147483648	2147483392

The most-significant binary digit is interpreted as the sign bit in any `Q` format number. Thus, in `Q15` format, the decimal point is placed immediately to the right of the sign bit. The fractional portion to the right of the sign bit is stored in regular two's complement format.

1.5 Compiler Requirements

When building the library source files or library-dependent modules it is assumed that the target is a Cadence processor implementing the Xtensa HiFi5 Audio Engine Instruction Set Architecture with VFPU/SFPU option.

1.6 Call Conventions

Library uses ANSI-C call conventions.

1.7 Overflow Control and Intermediate Data Format

If not especially noted, library does not check real dynamic range of input data so it is user's responsibility to select parameters and the scale of input data according to specific case. However, if possible library use saturated arithmetic to prevent overflows.
In the most fixed-point routines operating with summing of multiple elements (i.e. FIR, matrix multiplies, etc.), library stores intermediate values in 64-bit accumulators using Q16.47 fixed-point

representation thus protecting from the overflows in the intermediate stages. Floating point routines use single precision floating point format for storing intermediate data.
The user is expected to conform to the range requirements if specified and take care to restrict the input range in such a way that the outputs do not overflow.

1.8 Exceptions and Processor Control Registers

Except for some mathematical routines, compatible with IEEE-754 and C99 standards (see para 2.3), all library functions do not touch global `errno` variable and do not modify the FPU enabled bits. FPU flags may be set during the execution of the routines. It is up to the caller to decide how to proceed given the flags.

Example of use cases are:

- The caller could enable floating point control bits before calling functions. This would result in an external signal that indicates an exceptional condition has occurred. We expect the customer to use that signal to control an external interrupt – thus enabling an imprecise interrupt.
- The caller could zero the status flags before a function and check them when the function returns to see if any exceptional conditions occurred.

1.9 Special Numbers

The IEEE754 standard specifies some special values, and their representation: positive infinity (+∞ or `+Inf`), negative infinity (−∞ or `-Inf`), a negative zero (`-0`) distinct from ordinary ("positive") zero (`+0`), and "not a number" values (`NaN`s). In general, the following rules are applied:

- negative zero is treated as usual negative number
- the result of operations under `NaN` is `NaN`
- operations with infinity return `NaN` except for few routines which require to interpret only the sign of infinity
- If a result depends on several values (E.g. in filters and correlations), and one or more of them is `NaN` or `Inf`, the propagation of those special values is complicated. The library routines will propagate the value in a way that minimizes cycles and code size. A special value will still appear in the output.
- outputs for mathematical functions for special numbers on their inputs follows ISO/IEC 9899 if not explicitly mentioned

1.10 Endianess

Library supports little-endian mode.

1.11 Performance Issues

Real-time performance of all functions depends on fulfillment special restrictions applied to input/output arguments. Typically, for maximum performance, user have to use **aligned data arrays (on 16 byte boundary)** for storing input and output arguments, number of data should be **multiple of 2 or 4** and should be **greater than 4**. Specific requirements are given for each function in its API description.
Data alignment may be achieved by several methods:

- placing the data into special data section and make alignment at the link-time
- use `__attribute__((aligned(x)))` modifiers in the data declarations
- dynamically allocate arrays of slighter bigger size and align pointers[1]

Test examples use two last methods.

1.12 Object Model

Effective use of all HiFi5 core benefits requires specific processing and special data moves minimizing the overhead. That is why many functions are supplied with object-like interface simplifying real-time processing chain but requiring special initialization before processing. Besides, function wrapped by object-like interface use best possible alignment for data storage and may utilize HiFi5 core better in some cases.

Initialization normally done once at the initialization time and do not affect to the real-time performance. Sequence consists of three stages

- call `<obj>_alloc()` function with parameters that define the block size, filter length, etc. This function/macro returns the size of memory has to be allocated for object for that specific parameters
- allocate the memory somehow. It may be done dynamically if `<obj>_alloc()` function is used
- pass the pointer to allocated memory to the function `<obj>_init`. It cleans up that memory block, reorder filter coefficients appropriately, etc. and returns the handle to the object. This handle will be used later for data processing by this given object, .i.e. block filtering.

Here we denote the symbolic name of object as `<obj>`. For example, corresponding functions for block FIR filtering will be named as:

`bkfir_alloc()`	request the memory size for object
`bkfir_init()`	initialize the object
`bkfir_process()`	make filtering of block

1.13 Scratch Memory

Some functions require scratch memory area for saving temporary data during the execution. It can be shared between objects if they are called in the same processing thread. Scratch memory area must always be aligned on a 16-bytes boundary. For each algorithm that requires a scratch area, its minimum size (either in bytes or in 16-bit words) is specified in such a way that both static and dynamic allocation are possible. Most often the required scratch size is specified through a macro or function that takes a number of algorithm parameters. For example, one might use the following wrapping function to perform a autocorrelation (see **2.1.10)** with the scratch area being allocated on the stack:

```
#define N  80
int run_autocorrelation( int16_t * y, const int16_t *x )
{
  // Allocate the scratch memory. Macro returns required size in bytes
  int8_t scratch[FIR_ACORRA16X16_SCRATCH_SIZE(N) attribute__ ((aligned(16)));
  // Invoke the autocorrelation routine
  fir_acorra16x16  (scratch, y, x, N);
}
```

[1] xcc `malloc()` always returns pointer aligned on 128-bit boundary special additional alignment procedure is not required

1.14 Brief Function List

Vectorized version	Scalar version	Purpose	Reference
FIR filters and related functions			
bkfir		Block real FIR filter	2.1.1, 2.1.2
cxfir		Complex block FIR filter	2.1.3
firdec		Decimating block real FIR filter	2.1.4
firinterp		Interpolating block real FIR filter	2.1.5
fir_convol, cxfir_convol		Circular/linear convolution	2.1.6, 2.1.7
fir_xcorr		Circular/linear correlation	2.1.8, 2.1.9
fir_acorr		Circular/linear autocorrelation	2.1.10, 2.1.11
gccphat		Generalized PHAT Crosscorrelation	2.1.12
fir_blms		Blockwise Adaptive LMS algorithm	2.1.13
rppa, cppa, rpps, cpps		Polyphase analysis/synthesis filters	2.1.15, 2.1.16
IIR filters			
bqriir, bqciir		Biquad Real block IIR	2.2.1
latr		Lattice block Real IIR	2.2.2
kalmanupd		Kalman filter update	2.2.3
Mathematics			
vec_recip	scl_recip	Reciprocal on a vector of Q31 numbers	2.3.1
vec_divide	scl_divide	Division	2.3.2
vec_logn	scl_logn	Different kinds of logarithm	2.3.3
vec_log2	scl_log2		
vec_logn	scl_logn		
vec_recip	scl_recip	Reciprocal on a vector of Q31 numbers	2.3.1
vec_divide	scl_divide	Division	2.3.2
vec_logn	scl_logn	Different kinds of logarithm	2.3.3
vec_log2	scl_log2		
vec_logn	scl_logn		
vec_antilog2	scl_antilog2	Different kinds of antilogarithm	2.3.4
vec_antilog10	scl_antilog10		
vec_antilogn	scl_antilogn		
vec_pow		Power function	2.3.5
vec_sqrt	scl_sqrt	Square root	2.3.6
vec_rsqrt	scl_rsqrt	Reciprocal square root	2.3.7
vec_sine	scl_sine	Sine	2.3.8
vec_cosine	scl_cosine	Cosine	
vec_tan	scl_tan	Tangent	2.3.9
vec_atan, vec_atan2	scl_atan, scl_atan2	Arctangent	2.3.10, 2.3.11
vec_tanh	vec_tanh	Hyperbolic tangent	2.3.12
vec_sigmoid	scl_sigmoid	Sigmoid	2.3.13
vec_relu	scl_relu	Rectifier function (ReLU)	2.3.14
vec_softmax		Softmax	2.3.15
vec_int2float	scl_int2float	Integer to float conversion	2.3.16
vec_float2int	scl_float2int	Float to integer conversion	2.3.17
Complex Mathematics			
vec_complex2mag	scl_complex2mag	Complex magnitude	2.4.1
vec_complex2invmag	scl_complex2invmag	Reciprocal of complex magnitude	2.4.1
Vector operations			
vec_dot		Vector dot product	2.5.1, 2.6.4
vec_dot_batch		Batch Computation of Vector Dot Products	2.5.2

Vectorized version	Scalar version	Purpose	Reference
vec_add		Vector sum	2.5.2, **Ошибка! Источник ссылки не найден.**
vec_mul		Vector multiply, multiply-accumulate	2.6.2, 2.6.3
vec_power		Power of a vector	2.5.4
vec_shift vec_scale		Vector scaling with saturation	2.5.5
vec_bexp	scl_bexp	Common exponent	2.5.6
vec_min, vec_max		Find a maximum/minimum in a vector	2.5.7
Matrix operations			
mtx_mpy		Matrix multiply	2.7.1
mtx_vecmpy		Matrix by vector multiple	2.7.2
vec_vecmpy		Vector by vector multiply	2.7.3
cmtx_lrmpy		Three matrices product	2.7.4
mtx_transpose		Matrix transpose	2.7.5
Matrix Decomposition and Inversion Functions			
matchol		Cholesky decomposition, pseudoinversion	2.8.3, 2.8.4, 2.8.5, 2.8.6, 2.8.7, 2.8.8
mtx_gjelim		Gauss-Jordan matrix inversion	2.8.1, 2.8.2
Fitting/Interpolation			
vec_poly		Polynomial approximation	2.9.1
FFT/DCT			
fft_cplx		FFT on complex data	2.10.1
fft_real		FFT on real data	2.10.2
ifft_cplx		Inverse FFT on complex data	2.10.3
ifft_real		Inverse FFT forming real data	2.10.4
fft_cplx_ie		FFT on complex data with optimized memory usage	2.10.5
fft_real_ie		FFT on real data with optimized memory usage	2.10.6
ifft_cplx_ie		Inverse FFT on complex data with optimized memory usage	2.10.7
ifft_real_ie		Inverse FFT forming real data with optimized memory usage	2.10.8
fft_spectrum		Power spectrum	2.10.9
dct,mdct		Discrete cosine transform	2.10.10, 2.10.11
dct2d, idct2d		2D Discrete cosine transforms	2.10.12, 2.10.13
Mel frequency cepstrum coefficients functions			
LogMel		Mel frequency extraction	2.11.1
MFCC		MFCC	2.11.2
HTKdelta		HTK Delta Coefficients	2.11.3
Image processing			
imgpad		padding/cropping	2.12.1
imghist		histogram	2.12.2
imgnorm		normalization	2.12.3
imgconvert		RGB/YUV conversion	2.12.4
imginterleave, imgdeinterleave		interleaving/definterleaving	2.12.5
imgresize		resizing	2.12.6
imgrotate		rotation	2.12.7
imgfft, imgifft		2D FFT for images	2.12.8
Idenitification			
NatureDSP_Signal_get_library_version		Library Version Request	2.13.1

Vectorized version	Scalar version	Purpose	Reference
`NatureDSP_Signal_get_library_api_version`		Library API Version Request	2.13.2
`NatureDSP_Signal_isPresent`		Library API Capability Request	2.13.3

2 Reference

2.1 FIR Filters and Related Functions

FIR filtering APIs excepting correlation/convolution, autocorrelation and blockwise LMS algorithm require instantiation. In particular, filter objects encapsulate the delay line buffer, which is organized in such a way that advanced processor capabilities (e.g. circular data addressing) are efficiently utilized. When allocating and initializing a filter instance through `xfir_alloc()` and `xfir_init()` function calls, the user has to specify the length of filters and its coefficients. On the data processing stage the user application sequentially calls an `xfir_process()` function, providing it with a block of `N` input samples on each call. `xfir_process()` function updates the internal delay line with input samples, and computes `N` filter output samples, which are returned to the calling application via the output data buffer argument.

2.1.1 Block Real FIR Filter

Description

Computes a real FIR filter (direct-form) using IR stored in vector `h`. The real data input is stored in vector `x`. The filter output result is stored in vector `y`. The filter calculates `N` output samples using `M` coefficients and requires last `M-1` samples in the delay line which is updated in circular manner for each new sample. User has an option to set IR externally or copy from original location (i.e. from the slower constant memory). In the first case, user is responsible for right alignment, ordering and zero padding of filter coefficients – usually array is composed from zeroes (left padding), reverted IR and right zero padding.

Precision

6 variants available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs. Ordinary (single channel) variant and stereo
24x24p	use 24-bit data packing for internal delay line buffer and internal coefficients storage
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs. Ordinary (single channel) variant and stereo
32x32ep	32-bit data, 32-bit coefficients, 32-bit outputs, use 72-bit accumulators for intermediate computations.
f	floating point. Requires VFPU/SFPU core option. Ordinary (single channel) variant and stereo

Algorithm

$$y_n = \sum_{m=0}^{M-1} h_{M-1-m} x_{n+m}, n = \overline{0...N-1}$$

NOTE:
This is formal description of algorithm, in reality processing in done using circular buffers, so user application is not responsible for management of delay lines

Object allocation

```
size_t bkfir24x24p_alloc(int M, int extIR)
size_t bkfir16x16_alloc (int M, int extIR)
size_t bkfir32x16_alloc (int M, int extIR)
size_t bkfir32x32_alloc (int M, int extIR)
size_t bkfir32x32ep_alloc(int M, int extIR)
size_t bkfirf_alloc (int M, int extIR)
size_t stereo_bkfir16x16_alloc (int M, int extIR)
size_t stereo_bkfir32x32_alloc (int M, int extIR)
size_t stereo_bkfirf_alloc (int M, int extIR)
```

Type	Name	Size	Description
Input			

int	M		length of filter, should be a multiple of 4
int	extIR		if zero, IR is copied from original location, otherwise not but user should keep alignment, order of coefficients and zero padding requirements shown below

Returns: size of memory in bytes to be allocated

NOTE:
Approximate amount of requested memory is listed below

Function	Approximate memory requirements, bytes	
	extIR=0	extIR!=0
bkfir24x24p_alloc	80+M*6	56+M*3
bkfir16x16_alloc	72+M*4	56+M*2
bkfir32x16_alloc	72+M*6	64+M*4
bkfir32x32_alloc	72+M*8	48+M*4
bkfir32x32ep_alloc	72+M*8	48+M*4
bkfirf_alloc	72+M*8	48+M*4
stereo_bkfir16x16_alloc	72+M*8	56+M*4
stereo_bkfir32x32_alloc	72+M*16	48+M*8
stereo_bkfirf_alloc	72+M*16	48+M*8

Object initialization

```
bkfir24x24p_handle_t bkfir24x24p_init
            (void * objmem, int M, int extIR, const f24 * h)
bkfir16x16_handle_t bkfir16x16_init
            (void * objmem, int M, int extIR, const int16_t *  h)
bkfir32x16_handle_t bkfir32x16_init
            (void * objmem, int M, int extIR, const int16_t*  h)
bkfir32x32_handle_t bkfir32x32_init
            (void * objmem, int M, int extIR, const int32_t*  h)
bkfir32x32ep_handle_t bkfir32x32ep_init
            (void * objmem, int M, int extIR, const int32_t* h)
bkfirf_handle_t bkfirf_init
            (void * objmem, int M, int extIR, const float32_t*  h)
stereo_bkfir16x16_handle_t stereo_bkfir16x16_init
            (void * objmem, int M, int extIR,
             const int16_t *  hl, const int16_t *  hr)
stereo_bkfir32x32_handle_t stereo_bkfir32x32_init
            (void * objmem, int M, int extIR,
             const int32_t *  hl, const int32_t *  hr)
stereo_bkfirf_handle_t stereo_bkfirf_init
            (void * objmem, int M, int extIR,
             const float32_t *  hl, const float32_t *  hr)
```

Type	Name	Size	Description
Input			
void*	objmem		allocated memory block
f24, int16_t, int32_t, float32_t	h	M	filter coefficients; h[0] is to be multiplied with the newest sample
int16_t, int32_t, float32_t	hl	M	for stereo filters: filter coefficients for left channel
int16_t, int32_t, float32_t	hr	M	for stereo filters: filter coefficients for right channel
int	M		length of filter
int	extIR		if zero, IR is copied from original location, otherwise not but user should keep alignment, order of coefficients and zero padding requirements shown below

Returns: handle to the object

Alignment, ordering and zero padding for external IR (`extIR!=0`)

Function	Alignment, bytes	Left zero padding, bytes	Coefficient order	Right zero padding, bytes
bkfir24x24p_init	8	((-M&4)+5)*3	inverted	7
bkfir16x16_init	8	2	inverted	6
bkfir32x16_init (M>32)	8	10	inverted	6
bkfir32x16_init (M<=32)	8	2	inverted	6
bkfir32x32_init	8	4	inverted	12
bkfir32x32ep_init	8	4	inverted	12
bkfirf_init	8	0	direct	0
stereo_bkfir16x16_init	8	2	inverted	6
stereo_bkfir32x32_init	8	4	inverted	12
stereo_bkfirf_init	8	0	direct	0

Update the delay line and compute filter output

```
void bkfir24x24p_process( bkfir24x24p_handle_t handle,
                          f24* y,  const f24 * x, int N )
void bkfir16x16_process (
              bkfir16x16_handle_t  handle,
              int16_t *  y, const int16_t *  x, int N )
void bkfir32x16_process (
              bkfir32x16_handle_t  handle,
              int32_t *  y, const int32_t *  y, int N)
void bkfir32x32_process (
              bkfir32x32_handle_t  handle,
              int32_t *  y, const int32_t *  y, int N)
void bkfir32x32ep_process ( bkfir32x32ep_handle_t  handle,
                      int32_t * y, const int32_t * y, int N)
void bkfirf_process (
              bkfirf_handle_t  handle,
              float32_t * y, const float32_t * x, int N);
void stereo_bkfir16x16_process (
              stereo_bkfir16x16_handle_t  handle,
              int16_t *  y, const int16_t *  x, int N )
void stereo_bkfir32x32_process (
              stereo_bkfir32x32_handle_t  handle,
              int32_t *  y, const int32_t *  x, int N)
void stereo_bkfirf_process (
              stereo_bkfirf_handle_t  handle,
              float32_t * y, const float32_t * x, int N);
```

Type	Name	Size	Description
Input			
f24, int16_t, int32_t, float32_t	x	N*S	input samples
int	N		length of sample block
	S		1 for ordinary (single channel) filters, 2 - for stereo variant
Output			
f24, int16_t, int32_t, float32_t	y	N*S	output samples

Returns: none

Restrictions

`x,y` – should not overlap
`x,h` - aligned on a 16-bytes boundary
`N,M` - multiples of 4

2.1.2 Block Real FIR Filter with Arbitrary Parameters

Description

These functions implement FIR filter described in previous chapter with no limitation on size of data block,

alignment and length of impulse response for the cost of performance.

Precision

5 variants available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
32x32ep	32-bit data, 32-bit coefficients, 32-bit outputs, use 72-bit accumulators for intermediate computations
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$y_n = \sum_{m=0}^{M-1} h_{M-1-m} x_{n+m}, n = \overline{0...N-1}$$

NOTE:
This is formal description of algorithm, in reality processing in done using circular buffers, so user application is not responsible for management of delay lines

Object allocation

```
size_t bkfira16x16_alloc(int M)
size_t bkfira32x16_alloc(int M)
size_t bkfira32x32_alloc(int M)
size_t bkfira32x32ep_alloc(int M)
size_t bkfiraf_alloc(int M)
```

Type	Name	Size	Description
Input			
int	M		length of filter

Returns: size of memory in bytes to be allocated

Approximate amount of requested memory is listed below

Function	Approximate memory requirements, bytes
bkfira16x16_alloc	72+ M*4
bkfira32x16_alloc	72+ M*6
bkfira32x32_alloc	80+ M*8
bkfira32x32ep_alloc	80+ M*8
bkfiraf_alloc	80+ M*8

Object initialization

```
bkfira16x16_handle_t bkfira16x16_init
            (void * objmem, int M, const int16_t *  h)
bkfira32x16_handle_t bkfira32x16_init
            (void * objmem, int M, const int16_t*  h)
bkfira32x32_handle_t bkfira32x32_init
            (void * objmem, int M, const int32_t*  h)
bkfira32x32ep_handle_t bkfira32x32ep_init
            (void * objmem, int M, const int32_t*  h)
bkfiraf_handle_t bkfiraf_init
            (void * objmem, int M, const int16_t*  h)
```

Type	Name	Size	Description
Input			
void*	objmem		allocated memory block
int16_t, int32_t, float32_t	h	M	filter coefficients; h[0] is to be multiplied with the newest sample
int	M		length of filter

Returns: handle to the object

Update the delay line and compute filter output

```
void bkfira16x16_process (
                bkfira16x16_handle_t  handle,
                int16_t *  y, const int16_t *  x, int N );
void bkfira32x16_process (
                bkfira32x16_handle_t  handle,
                int32_t *  y, const int32_t *  y, int N);
void bkfira32x32_process (
                bkfira32x32_handle_t  handle,
                int32_t *  y, const int32_t *  y, int N);
void bkfira32x32ep_process (
                bkfira32x32ep_handle_t  handle,
                int32_t *  y, const int32_t *  y, int N);
void bkfiraf_process (
             bkfiraf_handle_t  handle,
             float32_t * y, const float32_t * x, int N);
```

Type	Name	Size	Description
Input			
`int16_t`, `int32_t`, `float32_t`	`x`	`N`	input samples
`int`	`N`		length of sample block
Output			
`int16_t`, `int32_t`, `float32_t`	`y`	`N`	output samples

Returns: none

Restrictions

`x`, `y` – should not overlap

2.1.3 Complex Block FIR Filter

Description

Computes a complex FIR filter (direct-form) using complex IR stored in vector `h`. The complex data input is stored in vector `x`. The filter output result is stored in vector `y`. The filter calculates `N` output samples using `M` coefficients, requires last `M-1` samples in the delay line which is updated in circular manner for each new sample. Real and imaginary parts are interleaved and real parts go first (at even indexes).
User has an option to set IR externally or copy from original location (i.e. from the slower constant memory). In the first case, user is responsible for right alignment, ordering and zero padding of filter coefficients – usually array is composed from zeroes (left padding), reverted IR and right zero padding.

Precision

5 variants available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
32x32ep	32-bit data, 32-bit coefficients, 32-bit outputs, use 72-bit accumulators for intermediate computations
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$y_n = \sum_{m=0}^{M-1} h_{M-1-m} x_{n+m}, n = \overline{0...N-1}$$

NOTE:
This is formal description of algorithm, in reality processing in done using circular buffers, so user application is not responsible for management of delay lines

Object allocation

```
size_t cxfir16x16_alloc(int M, int extIR)
size_t cxfir32x16_alloc(int M, int extIR)
size_t cxfir32x32_alloc(int M, int extIR)
size_t cxfir32x32ep_alloc(int M, int extIR)
size_t cxfirf_alloc(int M, int extIR)
```

Type	Name	Size	Description
Input			

int	M		length of filter

Returns: size of memory in bytes to be allocated
NOTE:
Approximate amount of requested memory is listed below

Function	Approximate memory requirements, bytes	
	extIR=0	extIR!=0
cxfir16x16_alloc	80+12*M	40+4*M
cxfir32x16_alloc	80+12*M	64+8*M
cxfir32x32_alloc	80+16*M	72+8*M
cxfir32x32ep_alloc	80+16*M	72+8*M
cxfirf_alloc	64+16*M	72+8*M

Object initialization

```
cxfir16x16_handle_t cxfir16x16_init(void * objmem,
                        int M, int extIR, const complex_fract16 *  h)
cxfir32x16_handle_t cxfir32x16_init(void * objmem,
                        int M, int extIR, const complex_fract16 *  h)
cxfir32x32_handle_t cxfir32x32_init(void * objmem,
                        int M, int extIR, const complex_fract32 *  h)
cxfir32x32ep_handle_t cxfir32x32ep_init(void * objmem,
                        int M, int extIR, const complex_fract32 *  h)
cxfirf_handle_t cxfirf_init(void * objmem,
                        int M, int extIR, const complex_float *  h)
```

Type	Name	Size	Description
Input			
void*	objmem		allocated memory block
complex_fract32, complex_fract16, complex_float	h	M	complex filter coefficients; `h[0]` is to be multiplied with the newest sample , Q31, Q15 or floating point
int	M		length of filter
int	extIR		if zero, IR is copied from original location, otherwise not but user should keep alignment, order of coefficients and zero padding requirements shown below

Returns: handle to the object

Alignment, ordering and zero padding for external IR (`extIR!=0`)

Function	Alignment, bytes	Left zero padding, bytes	Coefficient order	Right zero padding, bytes
cxfir16x16_alloc	8	2 before each copy	inverted: conjugated copy and (imaginary; real) copy at 4*(M+4) bytes offset	6 after each copy
cxfir32x16_init	8	4	inverted	4
cxfir32x32_init	8	0	inverted and conjugated	0
cxfir32x32ep_init	8	0	inverted and conjugated	0
cxfirf_init	8	0	direct	0

Update the delay line and compute filter output

```
void cxfir16x16_process(
            cxfir16x16_handle_t handle,
            complex_fract16 *  y,
            const complex_fract16*  x, int N );
void cxfir32x16_process(cxfir32x16_handle_t handle,
            complex_fract32 *  y,
            const complex_fract32 *  x, int N );
void cxfir32x32_process(cxfir32x32_handle_t handle,
            complex_fract32 *  y,
            const complex_fract32 *  x, int N );
void cxfir32x32ep_process(cxfir32x32ep_handle_t handle,
            complex_fract32 *  y,
            const complex_fract32 *  x, int N );
void cfirf_process (  cxfirf_handle_t handle,
              complex_float * y, const complex_float * x, int N);
```

Type	Name	Size	Description
Input			
complex_fract16, complex_fract32, complex_float	x	N	input samples , Q15, Q31 or floating point
int	N		length of sample block
Output			
complex_fract16, complex_fract32, complex_float	y	N	output samples , Q15, Q31 or floating point

Returns: none

Restrictions

`x,y` – should not overlap
`x,h` - aligned on a 16-bytes boundary
`N,M` - multiples of 4

2.1.4 Decimating Block Real FIR Filter

Description

Computes a real FIR filter (direct-form) with decimation using IR stored in vector `h`. The real data input is stored in vector `x`. The filter output result is stored in vector `y`. The filter calculates `N` output samples from `N*D` input samples using `M` coefficients, requires last `M-1` samples on the delay line and updated in circular manner for each new `D` samples.
NOTE:
To avoid aliasing IR should be synthesized in such a way to be narrower than input sample rate divided to 2D.

Precision

5 variants available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
32x32ep	32-bit data, 32-bit coefficients, 32-bit outputs, use 72-bit accumulators for intermediate computations
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$r_n = \sum_{m=0}^{M-1} h_{M-1-m} x_{D \cdot n+m}, n = \overline{0 \ldots N-1}$$

NOTE:
This is formal description of algorithm, in reality processing in done using circular buffers, so user application is not responsible for management of delay lines

Object allocation

```
size_t firdec16x16_alloc(int D, int M)
size_t firdec32x16_alloc(int D, int M)
size_t firdec32x32_alloc(int D, int M)
size_t firdec32x32ep_alloc(int D, int M)size_t firdecf_alloc    (int D, int M)
```

Type	Name	Size	Description
Input			
int	D		decimation factor
int	M		length of filter

Returns: size of memory in bytes to be allocated

NOTE:
Approximate amount of requested memory is listed below

Function	Approximate memory requirements, bytes
firdec32x16_alloc	40+(M+8*D)*4+(M+4)*2
firdec16x16_alloc	40+(M+8*D)*2+(M+4)*2
firdec32x32_alloc	40+(M+8*D)*4+(M+4)*4
firdec32x32ep_alloc	40+(M+8*D)*4+(M+4)*4
firdecf_alloc	40+(M+8*D)*4+(M+4)*4

Object initialization

```
firdec16x16_handle_t firdec16x16_init(void * objmem,
                        int D, int M, const int16_t *  h)
firdec32x16_handle_t firdec32x16_init(void * objmem,
                        int D, int M, const int16_t *  h)
firdec32x32_handle_t firdec32x32_init(void * objmem,
                        int D, int M, const int32_t *  h)
firdec32x32ep_handle_t firdec32x32ep_init(void * objmem,
                        int D, int M, const int32_t *  h)
firdecf_handle_t firdecf_init(void * objmem,
                        int D, int M, const float32_t *  h)
```

Type	Name	Size	Description
Input			
void*	objmem		allocated memory block
int32_t, int16_t, float32_t	h	M	filter coefficients; h[0] is to be multiplied with the newest sample, Q31, Q15 or floating point
int	D		decimation factor
int	M		length of filter

Returns: handle to the object

Update the delay line and compute decimator output

```
void firdec16x16_process(firdec16x16_handle_t handle,
                         int16_t *  y, const int16_t * x, int N );
void firdec32x16_process(firdec32x16_handle_t handle,
                         int32_t *  y, const int32_t * x, int N );
void firdec32x32_process(firdec32x32_handle_t handle,
                         int32_t *  y, const int32_t * x, int N );
void firdec32x32ep_process(firdec32x32ep_handle_t handle,
                         int32_t *  y, const int32_t * x, int N );
void firdecf_process (firdecf_handle_t handle,
                      float32_t * y, const float32_t * x, int N);
```

Type	Name	Size	Description
Input			
`int16_t`, `int32_t`, `float32_t`	`x`	`D*N`	input samples , Q15, Q31 or floating point
`int`	`N`		length of output sample block, should be a multiple of 8
Output			
`int16_t`, `int32_t`, `float32_t`	`y`	`N`	output samples, Q15, Q31 or floating point

Returns: none

Restrictions

`x,h,r` should not overlap
`x, h` - aligned on a 16-bytes boundary
`N` – multiple of 8
`D >1`

Conditions for optimum performance

`D` – 2, 3 or 4

2.1.5 Interpolating Block Real FIR Filter

Description

Computes a real FIR filter (direct-form) with interpolation using IR stored in vector `h`. The real data input is stored in vector `x`. The filter output result is stored in vector `y`. The filter calculates `N*D` output samples using `M*D` coefficients from `N` inputs. Delay line holds `M*D-1` last samples and updated in circular manner for each new sample.

Precision

5 variants available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
32x32ep	32-bit data, 32-bit coefficients, 32-bit outputs, use 72-bit accumulators for intermediate computations
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$y_{n\cdot D+d} = D\cdot\sum_{m=0}^{M-1} h_{D(M-1-m)+d}\, x_{n+m}, n=\overline{0..N-1}, d=\overline{0..D-1},$$

NOTE:
This is formal description of algorithm, in reality processing in done using circular buffers, so user application is not responsible for management of delay lines

Object allocation

```
size_t firinterp16x16_alloc(int D, int M)
size_t firinterp32x16_alloc(int D, int M)
size_t firinterp32x32_alloc(int D, int M)
size_t firinterp32x32ep_alloc(int D, int M)
size_t firinterpf_alloc    (int D, int M)
```

Type	Name	Size	Description
Input			
int	D		interpolation ratio
int	M		length of subfilter. Total length of filter is M*D

Returns: size of memory in bytes to be allocated

NOTE:

Approximate amount of requested memory is listed below

Function	Approximate memory requirements, bytes
firinterp16x16_alloc	40+(M+8)*2+(M+4)*D*2
firinterp32x16_alloc	40+(M+8)*4+(M+4)*D*2
firinterp32x32_alloc	40+(M+8)*4+(M+4)*D*4
firinterp32x32ep_alloc	40+(M+8)*4+(M+4)*D*4
firinterpf_alloc	40+(M+8)*4+(M+4)*D*4

Object initialization

```
firinterp16x16_handle_t firinterp16x16_init(void * objmem,
                        int D, int M, const int16_t*  h)
firinterp32x16_handle_t firinterp32x16_init(void * objmem,
                        int D, int M, const int16_t *  h)
firinterp32x32_handle_t firinterp32x32_init(void * objmem,
                        int D, int M, const int32_t *  h)
firinterp32x32ep_handle_t firinterp32x32ep_init(void * objmem,
                        int D, int M, const int32_t *  h)
firinterpf_handle_t firinterpf_init(void * objmem,
                        int D, int M, const float32_t *  h)
```

Type	Name	Size	Description
Input			
void*	objmem		allocated memory block
int32_t, int16_t, float32_t	h	M*D	filter coefficients; h[0] is to be multiplied with the newest sample,Q31, Q15 or floating point
int	D		interpolation ratio
int	M		length of subfilter. Total length of filter is M*D

Returns: handle to the object

Update the delay line and compute interpolator output

```
void firinterp16x16_process(firinterp16x16_handle_t handle,
                            int16_t *  y, const int16_t *  x, int N);
void firinterp32x16_process(firinterp32x16_handle_t handle,
                            int32_t*  y, const int32_t*  x, int N);
void firinterp32x32_process(firinterp32x32_handle_t handle,
                            int32_t*  y, const int32_t*  x, int N);
void firinterp32x32ep_process(firinterp32x32ep_handle_t handle,
                            int32_t*  y, const int32_t*  x, int N);
void firinterpf_process     (firinterpf_handle_t handle,
                            float32_t * y, const float32_t * x, int N);
```

Type	Name	Size	Description
Input			
int16_t, int32_t, float32_t	x	N	input samples, Q15, Q31 or floating point
int	N		length of input sample block
Output			
int16_t, int32_t, float32_t	y	N*D	output samples, Q15, Q31 or floating point

Returns: none

Restrictions

`x,h,y` should not overlap
`x,h` - aligned on a 16-bytes boundary
`M` - multiples of 4
`N` - multiples of 8
`D` should be >1

Conditions for optimum performance

`D` - 2, 3 or 4

2.1.6 Circular convolution

Description

Performs circular convolution between vectors `x` (of length `N`) and `y` (of length `M`) resulting in vector `r` of length `N`.
Two versions of these functions available: faster version (`fir_convol16x16, fir_convol32x16, fir_convol32x32, fir_convol32x32ep, cxfir_convol32x16, fir_convolf`) with some restrictions on input arguments and slower version (`fir_convola16x16, fir_convola32x16, fir_convola32x32, fir_convola32x32ep, cxfir_convola32x16, fir_convolaf`) for arbitrary arguments. In addition, these slower version implementations require scratch memory area.

Precision

5 variants available:

Type	Description
16x16	16x16-bit data, 16-bit outputs
32x16	32x16-bit data, 32-bit outputs (both real and complex)
32x32	32x32-bit data, 32-bit outputs
32x32ep	32-bit data, 32-bit outputs, use 72-bit accumulators for intermediate computations
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$r_k = \sum_{m=0}^{M-1} x_{\mathrm{mod}(k-m,N)} y_m, k = \overline{0\ldots(N-1)}$$

Prototype

```
void fir_convol16x16  ( int16_t *  r, const int16_t *  x,const int16_t *  y,
                        int N, int M);
void fir_convol32x16  ( int32_t *  r, const int32_t *  x, const int16_t *  y,
                        int N, int M);
void fir_convol32x32  ( int32_t *  r, const int32_t *  x, const int32_t *  y,
                        int N, int M);
void fir_convol32x32ep( int32_t *  r, const int32_t *  x, const int32_t *  y,
                        int N, int M);
void cxfir_convol32x16( complex_fract32 *  r,
                        const complex_fract32 *  x, const complex_fract16 *  y,
                        int N,int M);
void fir_convolf      ( float32_t *  r,
                        const float32_t *  x, const float32_t *  y,
                        int N, int M);

void fir_convola16x16 ( void *  s,
                        int16_t  *  r, const int16_t  *  x, const int16_t  *  y,
                        int N, int M);
void fir_convola32x16 ( void    *  s,
                        int32_t *  r, const int32_t *  x, const int16_t *  y,
                        int N, int M);
void fir_convola32x32 ( void    *  s,
                        int32_t *  r, const int32_t *  x, const int32_t *  y,
                        int N, int M);
void fir_convola32x32ep(void    *  s,
                        int32_t *  r, const int32_t *  x, const int32_t *  y,
                        int N, int M);
void cxfir_convola32x16(void    *  s,
                        complex_fract32 *  r,
                        const complex_fract32 *  x, const complex_fract16 *  y,
                        int N, int M);
void fir_convolaf      (void       *  s,
                        float32_t  *  r,
                        const float32_t  *  x, const float32_t  *  y,
                        int N, int M);
```

Arguments

Type	Name	Size	Description
Input			
int16_t, int32_t, complex_fract32, float32_t	x	N	input data (Q15, Q31 or floating point)
int16_t, complex_fract16, or float32_t	y	M	input data (Q31, Q15 or floating point)
int	N		length of x
int	M		length of y
Output			
int16_t, int32_t, complex_fract32, float32_t	r	N	output data, Q15, Q31 or floating point
Temporary			
void	s		Scratch memory, FIR_CONVOLA16X16_SCRATCH_SIZE(N, M) FIR_CONVOLA32X16_SCRATCH_SIZE(N, M) FIR_CONVOLA32X32_SCRATCH_SIZE(N, M) FIR_CONVOLA32X32EP_SCRATCH_SIZE(N, M) CXFIR_CONVOLA32X16_SCRATCH_SIZE(N,M) FIR_CONVOLAF_SCRATCH_SIZE(N, M) bytes

Returned value

none

Restrictions

For slow versions (fir_convola16x16, fir_convola32x16, fir_convola32x32, fir_convola32x32ep, cxfir_convola32x16, fir_convolaf):
x,y,r,s should not overlap
s should be aligned on 16-bytes boundary
N>=M-1

For fast versions (fir_convol16x16, fir_convol32x16, fir_convol32x32, fir_convol32x32ep, cxfir_convol32x16, fir_convolf):
x,y,r should not overlap
x,y,r should be aligned on 16-byte boundary
N,M – multiples of 4

2.1.7 Linear Convolution

Description

Functions perform linear convolution between vectors x (of length N) and y (of length M) resulting in vector r of length N+M-1.

Precision

2 variants available:

Type	Description
16x16	16x16-bit data, 16-bit outputs
32x32	32x32-bit data, 32-bit outputs

Algorithm

$$r_k = \sum_{j=\max(k-M+1,0)}^{\min(N-1,k)} x_j y_{k-j}, k = \overline{0\ldots(M+N-2)}$$

Prototype

```
void fir_lconvola16x16 (void    *  s,
                        int16_t  *  r,
                  const int16_t  *  x, const int16_t  *  y, int N, int M);
void fir_lconvola32x32 (void    *  s,
                        int32_t  *  r,
                  const int32_t  *  x, const int32_t  *  y, int N, int M);
```

Arguments

Type	Name	Size	Description
Input			
int16_t, int32_t	x	N	input data (Q15, Q31)
int16_t, int32_t	y	M	input data (Q31, Q15)
int	N		length of x
int	M		length of y
Output			
int16_t, int32_t	r	M+N-1	output data, Q15, Q31
Temporary			
void	s		Scratch memory, FIR_LCONVOLA16X16_SCRATCH_SIZE(N, M) FIR_LCONVOLA32X32_SCRATCH_SIZE(N, M) bytes

Returned value

none

Restrictions

x,y,r,s should not overlap
s should be aligned on 16-byte boundary
N>0, M>0
N>=M-1

2.1.8 Circular Correlation

Description

Estimates the circular cross-correlation between vectors x (of length N) and y (of length M) resulting in vector r of length N. It is a similar to correlation but y is read in opposite direction.
Two versions of these functions available: faster version (fir_xcorr16x16, fir_xcorr32x16, fir_xcorr32x32, fir_xcorr32x32ep, fir_xcorrf, cxfir_xcorrf) with some restrictions on input arguments and slower version (fir_xcorra16x16, fir_xcorra32x16, fir_xcorra32x32, fir_xcorra32x32ep, fir_xcorraf, cxfir_xcorraf) for arbitrary arguments. In addition, these slower version implementations require scratch memory area.

Precision

5 variants available:

Type	Description
16x16	16x16-bit data, 16-bit outputs
32x16	32x16-bit data, 32-bit outputs
32x32	32x32-bit data, 32-bit outputs (both real and complex data)
32x32ep	32-bit data, 32-bit outputs, use 72-bit accumulators for intermediate computations.
f	floating point (both real and complex data). Requires VFPU/SFPU core option

Algorithm

$$r_k = \sum_{m=0}^{M-1} x_{\mathrm{mod}(k+m,N)} y_m, k = \overline{0 \ldots (N-1)}$$

Prototype

```
void fir_xcorr16x16 ( int16_t *  r, const int16_t *  x, const int16_t *  y,
                      int N, int M);
void fir_xcorr32x16 ( int32_t *  r, const int32_t *  x, const int16_t *  y,
                      int N, nt M);
void fir_xcorr32x32 ( int32_t *  r, const int32_t *  x, const int32_t *  y,
                      int N, int M);
void fir_xcorr32x32ep(int32_t *  r, const int32_t *  x, const int32_t *  y,
                      int N, int M);
void cxfir_xcorr32x32 ( complex_fract32 *  r,
                      const complex_fract32 *  x, const complex_fract32 *  y,
                      int N, int M);
void fir_xcorrf      (float32_t *  r,
                      const float32_t *  x, const float32_t *  y,
                      int N, int M);
void cxfir_xcorrf    (complex_float *  r,
                      const complex_float *  x, const complex_float *  y,
                      int N, int M);
```

```
void fir_xcorra32x16 (void    *  s,
                      int32_t *  r, const int32_t *  x, const int16_t *  y,
                      int N, int M);
void fir_xcorra32x32 (void    *  s,
                      int32_t *  r, const int32_t *  x, const int32_t *  y,
                      int N, int M);
void cxfir_xcorra32x32 (void    *  s, complex_fract32 *  r,
                        const complex_fract32 *  x, const complex_fract32 *  y,
                        int N, int M);
void fir_xcorra32x32ep(void   *  s,
                      int32_t *  r, const int32_t *  x, const int32_t *  y,
                      int N, int M);
void fir_xcorraf     (void      *  s,
                      float32_t *  r, const float32_t * x, const float32_t * y,
                      int N, int M);
void cxfir_xcorraf   (void      *  s,
                      complex_float *  r,
                      const complex_float *  x, const complex_float *  y,
                      int N, int M);
```

Arguments

	Name	Size	Description
Input			
int16_t, int32_t, float32_t, complex_fract32, complex_float	x	N	input data (Q15, Q31 or floating point)
int16_t, float32_t, complex_fract32, complex_float	y	M	input data (Q31, Q15 or floating point)
int	N		length of x
int	M		length of y
Output			
int16_t, int32_t, float32_t, complex_fract32, complex_float	r	N	output data, Q15, Q31 or floating point
Temporary			
void	s		Scratch memory, FIR_XCORRA16X16_SCRATCH_SIZE(N, M) FIR_XCORRA32X16_SCRATCH_SIZE(N,M) FIR_XCORRA32X32_SCRATCH_SIZE(N, M) FIR_XCORRA32X32EP_SCRATCH_SIZE(N, M) FIR_XCORRAF_SCRATCH_SIZE(N, M) CXFIR_XCORRA32X32_SCRATCH_SIZE(N, M) CXFIR_XCORRAF_SCRATCH_SIZE(N, M) bytes

Returned value

none

Restrictions

For slow versions (fir_xcorra16x16, fir_xcorra32x16, fir_xcorra32x32ep, fir_xcorra32x32, fir_cxcorra32x32, fir_xcorraf, cxfir_xcorraf):
x,y,r,s should not overlap
s should be aligned on 16-byte boundary
N>=M-1

For fast versions (fir_xcorr16x16, fir_xcorr32x16, fir_xcorr32x32, fir_xcorr32x32ep, cxfir_xcorr32x32, fir_xcorrf, cxfir_xcorrf):
x,y,r should not overlap
x,y,r should be aligned on 16-byte boundary
N,M – multiples of 4

2.1.9 Linear Correlation

Description

Functions estimate the linear cross-correlation between vectors `x` (of length `N`) and `y` (of length `M`) resulting in vector `r` of length `N+M-1`. It is a similar to convolution but `y` is read in opposite direction.

Precision

2 variants available:

Type	Description
16x16	16x16-bit data, 16-bit outputs
32x32	32x32-bit data, 32-bit outputs

Algorithm

$$r_k = \sum_{j=\max(k-M+1,0)}^{\min(N-1,k)} x_j y^*_{M-1-(k-j)}, k = \overline{0\ldots(M+N-2)}$$

Prototype

```
void fir_lxcorra16x16 ( void    *  s,
                        int16_t  *  r,
                  const int16_t  *  x, const int16_t  *  y, int N, int M);
void fir_lxcorra32x32 ( void    *  s,
                        int32_t  *  r,
                  const int32_t  *  x, const int32_t  *  y, int N, int M);
```

Arguments

Type	Name	Size	Description
Input			
`int16_t, int32_t`	`x`	`N`	input data (Q15, Q31)
`int16_t, int32_t`	`y`	`M`	input data (Q31, Q15)
`int`	`N`		length of `x`
`int`	`M`		length of `y`
Output			
`int16_t, int32_t,`	`r`	`M+N-1`	output data, Q15, Q31
Temporary			
`void`	`s`		Scratch memory, `FIR_LXCORRA16X16_SCRATCH_SIZE(N, M)` `FIR_LXCORRA32X32_SCRATCH_SIZE(N, M)` bytes

Returned value

none

Restrictions

`x,y,r,s` should not overlap
`s` should be aligned on 16-byte boundary
`N>0, M>0`
`N>=M-1`

2.1.10 Circular Autocorrelation

Description

Estimates the auto-correlation of vector `x`. Returns autocorrelation of length `N`.
Two versions of these functions available: faster version (`fir_acorr16x16, fir_acorr32x32, fir_acorr32x32ep, fir_acorrf`) with some restrictions on input arguments and slower version (`fir_acorra16x16, fir_acorra32x32, fir_acorra32x32ep, fir_acorraf`) for arbitrary arguments. In addition, this slower version implementations require scratch memory area.

Precision

4 variants available:

Type	Description
16x16	16-bit data, 16-bit outputs
32x32	32-bit data, 32-bit outputs
32x32ep	32-bit data, 32-bit outputs, use 72-bit accumulators for intermediate computations.
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$r_k = \sum_{n=0}^{N-1} x_{\mathrm{mod}(n+k,N)} x_n, k = \overline{0\ldots(N-1)}$$

Prototype

```
void fir_acorr16x16  ( int16_t  *  r, const int16_t  *  x, int N);
void fir_acorr32x32  ( int32_t  *  r, const int32_t  *  x, int N);
void fir_acorr32x32ep( int32_t  *  r, const int32_t  *  x, int N);
void fir_acorrf      ( float32_t*  r, const float32_t*  x, int N);

void fir_acorra16x16  (void* s, int16_t  *  r, const int16_t  *  x, int N);
void fir_acorra32x32  (void* s, int32_t  *  r, const int32_t  *  x, int N);
void fir_acorra32x32ep(void* s, int32_t  *  r, const int32_t  *  x, int N);
void fir_acorraf      (void* s, float32_t*  r, const float32_t*  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int16_t, int32_t or float32_t`	`x`	`N`	input data (Q15, Q31 or floating point)
`int`	`N`		length of `x`
Output			
`int16_t, int32_t or float32_t`	`r`	`N`	output data, Q15, Q31 or floating point
Temporary			
`void`	`s`		Scratch memory, `FIR_ACORRA16X16_SCRATCH_SIZE( N )` `FIR_ACORRA32X32_SCRATCH_SIZE( N )` `FIR_ACORRAF_SCRATCH_SIZE( N )` bytes

Returned value

none

Restrictions

For slow versions (`fir_acorr16x16, fir_acorr32x32, fir_acorr32x32ep, fir_acorrf`):
`x,r,s` should not overlap
`N` - must be non-zero
`s` - aligned on an 16-bytes boundary
For fast versions (`fir_acorra16x16, fir_acorra32x32, fir_acorra32x32ep, fir_acorraf`):
`x,r` should not overlap
`x,r` should be aligned on 16-byte boundary
`N` – non-zero multiple of 4

2.1.11 Linear Autocorrelation

Description

Functions estimate the linear auto-correlation of vector `x`. Returns autocorrelation of length `N`.

Precision

2 versions available:

Type	Description
16x16	16-bit data, 16-bit outputs
32x32	32-bit data, 32-bit outputs

Algorithm

$$r_k = \sum_{n=0}^{N-k-1} x_{n+k} x_n, k = \overline{0 \ldots (N-1)}$$

Prototype

```
void fir_lacorra16x16 (void* s, int16_t  *  r, const int16_t  *  x, int N);
void fir_lacorra32x32 (void* s, int32_t  *  r, const int32_t  *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
int16_t, int32_t	x	N	input data (Q15, Q31)
int	N		length of x
Output			
int16_t, int32_t	r	N	output data, Q15, Q31
Temporary			
void	s		Scratch memory, FIR_LACORRA16X16_SCRATCH_SIZE(N) FIR_LACORRA32X32_SCRATCH_SIZE(N) bytes

Returned value

none

Restrictions

x,r,s should not overlap
N>0
s - aligned on an 16-bytes boundary

2.1.12 Generalized Cross-Correlation Phase Transform (GCC-PHAT)

Description

Estimate the generalized cross-correlation of input vectors x and y to define the time-shift between them:
NOTE:
GCC-PHAT implementation performs a few FFT transforms of size N rounded up to the next power of 2. That is, these functions are utilized most efficiently when the input argument N is already a power of 2

Precision

2 versions available:

Type	Description
32x32	real 32-bit input/output data
f	real floating point input/output data

Algorithm

$$Z = FFT(x) \cdot \left(FFT(y)\right)^*$$

$$z = FFT^{-1}\left(\frac{Z}{|Z|}\right)$$

Prototype

```
int  gccphat32x32 (void* pScr, int32_t *z,
                   const int32_t *x, const int32_t *y, int N);
void gccphatf     (void* pScr, float32_t *z,
                   const float32_t *x, const float32_t *y, int N);
```

Scratch allocation functions

```
size_t gccphat32x32_getScratchSize (int N);
size_t gccphatf_getScratchSize     (int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t, float32_t	x	N	input data (Q31, floating point)
int32_t, float32_t	y	N	input data (Q31, floating point)
int	N		input/output vectors size
Output			
int32_t, float32_t	z	N	GCC-PHAT results vector, Q31 for the fixed-point variant
Temporary			
void	pScr		scratch memory area of size specified by the corresponding scratch allocation function (in bytes).

Returned value

Fixed-point variant function returns the sum of bi-directional right shift amounts applied throughout the inverse FFT transform.

Restrictions

x,y,z must not overlap and must be aligned by 16-bytes
pScr must be aligned by 16-bytes
N multiple of 32, 64<=N<=320

2.1.13 Blockwise Adaptive LMS Algorithm for Real Data

Description

Blockwise LMS algorithm performs filtering of reference samples `x[N+M-1]`, computation of error `e[N]` over a block of input samples `r[N]` and makes blockwise update of IR to minimize the error output. Algorithm includes FIR filtering, calculation of correlation between the error output `e[N]` and reference signal `x[N+M-1]` and IR taps update based on that correlation.

NOTES:

1. The algorithm must be provided with the normalization factor, which isthe power of the reference signal times `N` - the number of samples in a data block. This can be calculated using the `vec_power32x32()` or `vec_power16x16()` function. In order to avoid the saturation of the normalization factor, it may be biased, i.e. shifted to the right. If it's the case, then the adaptation coefficient must be also shifted to the right by the same number of bit positions.
2. This algorithm consumes less CPU cycles per block than single sample algorithm at similar convergence rate.
3. Right selection of `N` depends on the change rate of impulse response: on static or slow varying channels convergence rate depends on selected `mu` and `M`, but not on `N`.
4. 16x16 routine may converge slower on small errors due to roundoff errors. In that cases, 16x32 routine will give better results although convergence rate on bigger errors is the same
5. Terms near-end and far-end come from echo cancellation theory where the LMS algorithm is used widely. For echo cancellation the term far-end means the output of speakerphone (far end designates that the origin of it is somewhere outside say came from the remote speaker). The near-end is a signal from the local microphone representing a sum of the echo, speech of local speaker and the noise. The LMS is used to estimate the equivalent impulse response of the echopath further compensation and removal of the echo from the near-end signal.

Precision

5 variants available:

Type	Description
16x16	16-bit coefficients, 16-bit data, 16-bit output
16x32	32-bit coefficients, 16-bit data, 16-bit output
32x32	32-bit coefficients, 32-bit data, 32-bit output, complex and real
32x32ep	32-bit data, 32-bit coefficients, 32-bit outputs, use 72-bit accumulators for intermediate computations
f	floating point, both complex and real data. Requires VFPU/SFPU core option

Algorithm

$$b = \frac{\mu}{norm}$$

$$e_n = r_n - \sum_{m=0}^{M-1} h_{M-1-m} x_{m+n}, n = \overline{0..N-1}$$

$$h_{M-1-m} = h_{M-1-m} + b \cdot \sum_{n=0}^{N-1} e_n^* x_{n+m}, m = \overline{0..M-1}$$

Prototype

```
void fir_blms16x16 (  int16_t*  e, int16_t *  h,
                const int16_t *  r,
                const int16_t *  x,
                int16_t norm, int16_t   mu,
                int   N,   int   M);
void fir_blms16x32 (  int32_t *  e, int32_t *  h,
                const int16_t *  r,
                const int16_t *  x,
                int32_t   norm,int16_t   mu,
                int       N,   int       M);
void fir_blms32x32 (  int32_t *  e, int32_t *  h,
                const int32_t *  r,
                const int32_t *  x,
                int32_t   norm, int32_t mu,
                int       N,   int       M);
void cxfir_blms32x32 (complex_fract32 *  e, complex_fract32 *  h,
                const complex_fract32 *  r,
                const complex_fract32 *  x,
                int32_t   norm, int32_t mu,
                int       N,   int       M);

void fir_blms32x32ep(  int32_t *  e, int32_t *  h,
                const int32_t *  r,
                const int32_t *  x,
                int32_t   norm, int32_t mu,
                int       N,   int       M);
void fir_blmsf      ( float32_t * e, float32_t * h, const float32_t * r,
                const float32_t * x,
                float32_t norm, float32_t mu,
                int          N, int       M );
void cxfir_blmsf    ( complex_float * e, complex_float * h,
                const complex_float * r,
                const complex_float * x,
                float32_t norm, float32_t mu,
                int          N, int       M );
```

Arguments

Type	Name	Size	Description
Input			
int16_t, int32_t, complex_fract32, float32_t, complex_float	h	M	impulse response, Q15, Q31 or floating point
int16_t, int32_t complex_fract32, float32_t, complex_float	r	N	input (near end) data vector. First in time value is in r[0], Q31, Q15 or floating point
int16_t, int32_t, complex_fract32, float32_t, complex_float	x	N+M-1	reference (far end) data vector. First in time value is in x[0], Q31, Q15 or floating point
int16_t, int32_t, float32_t	norm		normalization factor: power of signal multiplied by N, Q31, Q15 or floating point
int16_t int32_t, float32_t	mu		adaptation coefficient in Q31, Q15 or floating point (LMS step)
int	N		length of data block
int	M		length of h
Output			
int16_t, int32_t, complex_fract32, float32_t, complex_float	e	N	estimated error, Q31, Q15 or floating point
int16_t, int32_t, float32_t, complex_float	h	M	updated impulse response, Q31, Q15 or floating point

Returned value none

Restrictions
`h,x,r,e` – should not overlap
`x,e,h,r` - aligned on a 16-bytes boundary
`N,M` - multiples of 8

2.1.14 2D Convolution

Description The `conv2d()` functions compute the two-dimensional convolution of input matrix `x[M][N]` and `y[P][Q]` and store the result in matrix `z[M+P-1][N+Q-1]`
Additional parameter `rsh` allows to control fixed point representation of output data.
Two versions of functions available:
- generic version with _gen_ suffix. These functions work with arbitrary arguments.
- fast version with no _gen_ suffix. These functions expose some additional restrictions on argument.

Precision 5 variants available:

Type	Description
8x8	8-bit coefficients, 8-bit data, 8-bit output, Q7
8x16	8-bit coefficients Q7, 16-bit data, 16-bit output, Q15
16x16	16-bit coefficients, 16-bit data, 16-bit output, Q15
f	single precision floating point
fp16	half precision floating point

Algorithm

$$y_{i,j} = 2^{-rsh} \sum_{m=\max(i-P+1,0)}^{\min(i,M-1)} \sum_{n=\max(j-Q+1,0)}^{\min(j,N-1)} x_{m,n} \cdot y_{j-m,i-n} \text{ where } i = \overline{0..M+P-2},\ j = \overline{0..N+Q-2}$$

Prototypes

```
void conv2d_8x8 (void *pScr, int8_t *z, const int8_t * x, const int8_t * y,
                 int rsh, int P, int Q);
void conv2d_8x16(void *pScr, int16_t *z, const int8_t * x, const int16_t * y,
                 int rsh, int P, int Q);
void conv2d_16x16(void *pScr, int16_t *z, const int16_t * x, const int16_t * y,
                  int rsh, int P, int Q);
void conv2df(void *pScr, float32_t *z, const float32_t * x, const float32_t * y,
                  int P, int Q);
void conv2d_fp16(void *pScr, float16_t *z, const float16_t * x, const float16_t
                  *y, int P, int Q);
```

Function	API	Scratch allocation function	Dimensions MxN
Generic version:			
conv2d_gen_3x3_8x8	conv2d_8x8	conv2d_gen_3x3_8x8_getScratchSize	3x3
conv2d_gen_5x5_8x8	conv2d_8x8	conv2d_gen_5x5_8x8_getScratchSize	5x5
conv2d_gen_11x7_8x8	conv2d_8x8	conv2d_gen_11x7_8x8_getScratchSize	11x7
conv2d_gen_3x3_8x16	conv2d_8x16	conv2d_gen_3x3_8x16_getScratchSize	3x3
conv2d_gen_5x5_8x16	conv2d_8x16	conv2d_gen_5x5_8x16_getScratchSize	5x5
conv2d_gen_11x7_8x16	conv2d_8x16	conv2d_gen_11x7_8x16_getScratchSize	11x7
conv2d_gen_3x3_16x16	conv2d_16x16	conv2d_gen_3x3_16x16_getScratchSize	3x3
conv2d_gen_5x5_16x16	conv2d_16x16	conv2d_gen_5x5_16x16_getScratchSize	5x5
conv2d_gen_11x7_16x16	conv2d_16x16	conv2d_gen_11x7_16x16_getScratchSize	11x7
conv2d_gen_3x3f	conv2df	conv2d_gen_3x3f_getScratchSize	3x3
conv2d_gen_5x5f	conv2df	conv2d_gen_5x5f_getScratchSize	5x5
conv2d_gen_11x7f	conv2df	conv2d_gen_11x7f_getScratchSize	11x7
conv2d_gen_3x3_fp16	conv2d_fp16	conv2d_gen_3x3_fp16_getScratchSize	3x3
conv2d_gen_5x5_fp16	conv2d_fp16	conv2d_gen_5x5_fp16_getScratchSize	5x5
conv2d_gen_11x7_fp16	conv2d_fp16	conv2d_gen_11x7_fp16_getScratchSize	11x7
Fast version:			
conv2d_3x3_8x8	conv2d_8x8	conv2d_3x3_8x8_getScratchSize	3x3
conv2d_5x5_8x8	conv2d_8x8	conv2d_5x5_8x8_getScratchSize	5x5

conv2d_11x7_8x8	conv2d_8x8	conv2d_11x7_8x8_getScratchSize	11x7
conv2d_3x3_8x16	conv2d_8x16	conv2d_3x3_8x16_getScratchSize	3x3
conv2d_5x5_8x16	conv2d_8x16	conv2d_5x5_8x16_getScratchSize	5x5
conv2d_11x7_8x16	conv2d_8x16	conv2d_11x7_8x16_getScratchSize	11x7
conv2d_3x3_16x16	conv2d_16x16	conv2d_3x3_16x16_getScratchSize	3x3
conv2d_5x5_16x16	conv2d_16x16	conv2d_5x5_16x16_getScratchSize	5x5
conv2d_11x7_16x16	conv2d_16x16	conv2d_11x7_16x16_getScratchSize	11x7
conv2d_3x3f	conv2df	conv2d_3x3f_getScratchSize	3x3
conv2d_5x5f	conv2df	conv2d_5x5f_getScratchSize	5x5
conv2d_11x7f	conv2df	conv2d_11x7f_getScratchSize	11x7
conv2d_3x3_fp16	conv2d_fp16	conv2d_3x3_fp16_getScratchSize	3x3
conv2d_5x5_fp16	conv2d_fp16	conv2d_5x5_fp16_getScratchSize	5x5
conv2d_11x7_fp16	conv2d_fp16	conv2d_11x7_fp16_getScratchSize	11x7

Arguments

Type	Name	Size	Description
Input			
int8_t, int16_t, float32_t, float16_t	x	[M][N]	input data, Q15, Q7 or floating point
int8_t, int16_t, float32_t, float16_t	y	[P][Q]	input data, Q15, Q7 or floating point
int	rsh		additional right shift (for fixed point API only)
	M		number of rows in the matrix x
	N		number of columns in the matrix x
int	P		number of rows in the matrix y
int	Q		number of columns in the matrix y
Temporary:			
pScr			scratch data. Should have size in bytes at least as requested by corresponding scratch allocation function
Output			
int8_t, int16_t, float32_t, float16_t	z	[M+P-1]*[N+Q-1]	output data, Q(7-rsh), Q(15-rsh) , or floating point

Returned value None

Restrictions

For fast version

x,y,z	should not overlap
x,y,z, pScr	aligned on a 16-bytes boundary
M,N,P,Q	must be positive
P, Q	must be bigger than zero and multiplies of 8

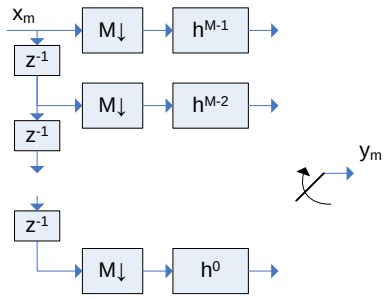
For regular version

x,y,z	should not overlap
pScr	aligned on a 16-bytes boundary
M,N,P,Q	must be positive
P, Q	must be bigger than zero

2.1.15 Polyphase Analysis FIR Filter

Description Polyphase analysis filters perform filtering of input sample block x[M]. Each input sample is passed through a dedicated subfilter (altogether M subfilters, each of length N). Output samples are organized into

type-1 polyphase decomposition frame `y[M]`, that is the output sample of 0th subfilter is stored to `y[0]`, output sample of 1st subfilter is stored to `y[1]` and so forth.



Precision

2 variants available:

Type	Description
32x32	32-bit data, 32-bit coefficients, 32-bit outputs, real and complex
f	floating point. Requires VFPU/SFPU core option, real and complex

Algorithm

$$y_m = \sum_{n=0}^{N-1} h_n^m x_{m-M\cdot n}, \text{where} h_n^m = h[M\cdot n + m], n = \overline{0...N-1}, m = \overline{0...M-1}$$

NOTE:
This is formal description of algorithm, in reality processing in done using circular buffers, so user application is not responsible for management of delay lines.

Object allocation

```
size_t rppafir32x32_alloc (int M, int N)
size_t cppafir32x32_alloc (int M, int N)
size_t rppafirf_alloc     (int M, int N)
size_t cppafirf_alloc     (int M, int N)
```

Type	Name	Size	Description
Input			
`int`	`M`		length of sample block
`int`	`N`		length of filter

Returns: size of memory in bytes to be allocated

Object initialization

```
rppafir32x32_handle_t rppafir32x32_init
            (void * objmem, int M, int N, const int32_t * h)
cppafir32x32_handle_t cppafir32x32_init
            (void * objmem, int M, int N, const int32_t * h)
rppaf_handle_t rppafirf_init
            (void * objmem, int M, int N, const float32_t * h)
cppaf_handle_t cppafirf_init
            (void * objmem, int M, int N, const float32_t * h)
```

Type	Name	Size	Description
Input			
`void*`	`objmem`		allocated memory block
`int32_t`, `float32_t`	`h`	`M*N`	filter coefficients for all `M` filters
`int`	`M`		length of sample block
`int`	`N`		length of filter

Returns: handle to the object

Update the delay line and compute filter output

```
void rppafir32x32_process(rppafir32x32_handle_t handle,
                          int32_t* y,  const int32_t * x);
void cppafir32x32_process(cppafir32x32_handle_t handle,
                          complex_fract32* y,  const complex_fract32 * x);
void rppafirf_process(rppafirf_handle_t handle,
                          float32_t* y,  const float32_t * x);
void cppafirf_process(cppafirf_handle_t handle,
                          complex_float* y,  complex_float * x);
```

Type	Name	Size	Description
Input			
int32_t, complex_fract32, float32_t, complex_float	x	M	input samples
Output			
int32_t, complex_fract32, float32_t, complex_float	y	M	output samples

Returns: none

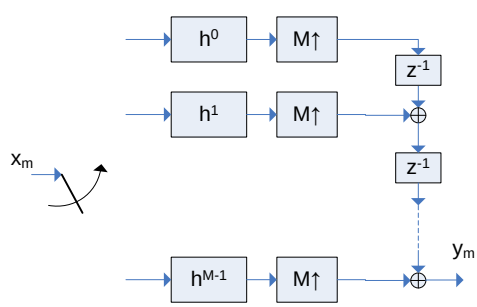
Restrictions

x,y	should not overlap
x,h	aligned on a 16-bytes boundary
N	4..24, multiplies of 2
M	32...640, multiplies of 32

2.1.16 Polyphase Synthesis FIR Filter

Description

Polyphase synthesis filter reconstructs a block of `M` samples `y[M]` from type-1 polyphase decomposition frame `x[M]`. First, each sample of the input frame `x[M]` is passed through a dedicated subfilter (altogether `M` subfilters, each of length `N`). After that, output samples of every subfilter are interleaved such that the first-in-time sample of the reconstructed block is stored to `y[0]`, the next-in-time sample – to `y[1]`, and so forth.



Normally, polyphase decomposition frames `x[M]` come from IFFT routines which may have scaled data to prevent overflow in the case of fixed point data. Input argument `lsh` accommodates the need to scale the reconstructed signal back to the desired representation.

Precision

2 variants available:

Type	Description
32x32	32-bit data, 32-bit coefficients, 32-bit outputs, real and complex
f	floating point. Requires VFPU/SFPU core option, real and complex

Algorithm

intermediate values:

$$y_m = 2^{lsh} \cdot \sum_{n=0}^{N-1} h_n^{M-1-m} x_{m-M\cdot n}, \text{where} h_n^m = h[M \cdot n + m], n = \overline{0..N-1}, m = \overline{0..M-1}$$

NOTE:
This is formal description of algorithm, in reality processing in done using circular buffers, so user application is not responsible for management of delay lines.

Object allocation

```
size_t rppsfir32x32_alloc  (int M, int N)
size_t cppsfir32x32_alloc  (int M, int N)
size_t rppsfirf_alloc      (int M, int N)
size_t cppsfirf_alloc      (int M, int N)
```

Type	Name	Size	Description
Input			
int	M		length of sample block
int	N		length of filter

Returns: size of memory in bytes to be allocated

Object initialization

```
rppsfir32x32_handle_t rppsfir32x32_init
           (void * objmem, int M, int N, const int32_t * h)
cppsfir32x32_handle_t cppsfir32x32_init
           (void * objmem, int M, int N, const int32_t * h)
rppsfirf_handle_t rppsfirf_init
           (void * objmem, int M, int N, const float32_t * h)
cppsfirf_handle_t cppsfirf_init
           (void * objmem, int M, int N, const float32_t * h)
```

Type	Name	Size	Description
Input			
void*	objmem		allocated memory block
int32_t, float32_t	h	M*N	filter coefficients for all M filters
int	M		length of sample block
int	N		length of filter

Returns: handle to the object

Update the delay line and compute filter output

```
void rppsfir32x32_process(rppsfir32x32_handle_t handle,
                     int32_t* y,  const int32_t * x, int lsh );
void cppsfir32x32_process(cppsfir32x32_handle_t handle,
                     complex_fract32* y,  const complex_fract32 * x, int lsh );
void rppsfirf_process(rppsfirf handle_t handle,
                     float32_t* y,  const float32_t * x);
void cppsfirf_process(cppsfirf handle_t handle,
                     complex_float* y,  complex_float * x);
```

Type	Name	Size	Description
Input			
int32_t, complex_fract32, float32_t, complex_float	x	M	input samples
int	lsh		scale factor (positive for left shift, negative for right shift) for scaling resulted output. For fixed point API only.
Output			
int32_t, complex_fract32, float32_t, complex_float	y	M	output samples

Returns: none

Restrictions

x,y	should not overlap
x,h	aligned on a 16-bytes boundary
N	4..24, multiplies of 2
M	32...640, multiplies of 32

2.2 IIR filters

2.2.1 Bi-quad Real Block IIR

Description

Computes a real IIR filter (cascaded IIR direct form I or II using 5 coefficients per bi-quad + gain term) . Input data are stored in vector `x`. Filter output samples are stored in vector `r`. The filter calculates `N` output samples using SOS and G matrices.
Due to the performance reasons, IIR biquad filters may introduce additional algorithmic delay of several samples. So, all the filters are available in 2 implementations

- first fastest variant use some additional delay lines for better parallelization. Amount of above algorithmic delay depends on the filter structure, number of sections, type of the data and might be requested by `xxx_groupDelay()` API

another one (with suffix `nd`) selects slower (especially with the big number of biquads) delayless algorithms. For these functions `xxx_groupDelay()` API functions return zero.
NOTE:

- Bi-quad coefficients may be derived from standard SOS and G matrices generated by MATLAB. However, typically biquad stages have big peaks in their step response which may cause undesirable overflows at the intermediate outputs. To avoid that the additional scale factors `coef_g[M]` may be applied. These per-section scale factors may require some tuning to find a compromise between quantization noise and possible overflows. Output of the last section is directed to an additional multiplier, with the gain factor being a power of two, either negative or non-negative. It is specified through the total gain shift amount parameter `gain` of each filter initialization function
- 16x16 filters may suffer more from accumulation of the roundoff errors, so filters should be properly designed to match noise requirements

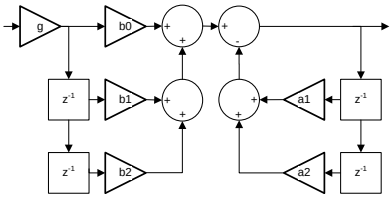
Precision

4 variants available:

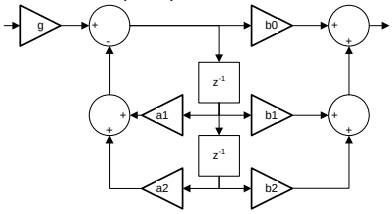
Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit intermediate stage outputs (DF1, DF1 stereo, DF II form)
32x16	32-bit data, 16-bit coefficients, 32-bit intermediate stage outputs (DF1, DF1 stereo, DF II form)
32x32	32-bit data, 32-bit coefficients, 32-bit intermediate stage outputs (DF I, DF1 stereo, DF II form)
f	floating point (DF I, DF1 stereo, DF II and DF IIt). Requires VFPU/SFPU core option

Algorithm

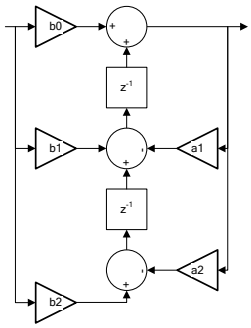
A block of N real input samples is sequentially passed through M bi-quad sections. There are two options for the implementation structure of a single section:
Direct Form I (DFI)



Direct Form II (DFII)



Direct Form II transposed (DF IIt)



Object allocation

```
size_t bqriir16x16_df1_alloc(int M)
size_t bqriir16x16_df2_alloc(int M)
size_t bqriir32x16_df1_alloc(int M)
size_t bqriir32x16_df2_alloc(int M)
size_t bqriir32x32_df1_alloc(int M)
size_t bqriir32x32_df2_alloc(int M)
size_t bqriirf_df1_alloc(int M)
size_t bqriirf_df2_alloc(int M)
size_t bqriirf_df2t_alloc(int M)
size_t bqciirf_df1_alloc(int M)
size_t stereo_bqriir16x16_df1_alloc(int M)
size_t stereo_bqriir32x16_df1_alloc(int M)
size_t stereo_bqriir32x32_df1_alloc(int M)
size_t stereo_bqriirf_df1_alloc    (int M)
```

For delayless filters:

```
size_t bqriir16x16_df1_nd_alloc(int M);
size_t bqriir16x16_df2_nd_alloc(int M);
size_t bqriir32x16_df1_nd_alloc(int M);
size_t bqriir32x16_df2_nd_alloc(int M);
size_t bqriir32x32_df1_nd_alloc(int M);
size_t bqriir32x32_df2_nd_alloc(int M);
size_t bqriirf_df1_nd_alloc(int M);
size_t bqriirf_df2_nd_alloc(int M);
size_t bqriirf_df2t_nd_alloc(int M);
size_t bqciirf_df1_nd_alloc(int M);
size_t stereo_bqriir16x16_df1_nd_alloc(int M);
size_t stereo_bqriir32x16_df1_nd_alloc(int M);
size_t stereo_bqriir32x32_df1_nd_alloc(int M);
size_t stereo_bqriirf_df1_nd_alloc(int M);
```

Type	Name	Size	Description
Input			
`int`	`M`		number of bi-quad sections

Returns: size of memory in bytes to be allocated

Object initialization

```
bqriir16x16_df1_handle_t bqriir16x16_df1_init(void * objmem, int M,
           const int16_t * coef_sos, const int16_t * coef_g, int16_t gain );
bqriir16x16_df2_handle_t bqriir16x16_df2_init(void * objmem, int M,
           const int16_t * coef_sos, const int16_t * coef_g,  int16_t gain);
bqriir32x16_df1_handle_t bqriir32x16_df1_init(void * objmem, int M,
           const int16_t * coef_sos, const int16_t * coef_g, int16_t gain);
bqriir32x16_df2_handle_t bqriir32x16_df2_init(void * objmem, int M,
           const int16_t * coef_sos, const int16_t * coef_g, int16_t gain);
bqriir32x32_df1_handle_t bqriir32x32_df1_init(void * objmem, int M,
           const int32_t * coef_sos, const int16_t * coef_g, int16_t gain)
bqriir32x32_df2_handle_t bqriir32x32_df2_init(void * objmem, int M,
           const int32_t * coef_sos, const int16_t * coef_g, int16_t gain)
bqriirf_df1_handle_t bqriirf_df1_init(void * objmem, int M,
           const float32_t* coef_sos, int16_t gain );
bqriirf_df2_handle_t bqriirf_df2_init(void * objmem, int M,
           const float32_t * coef_sos, int16_t gain);
bqriirf_df2t_handle_t bqriirf_df2t_init(void * objmem, int M,
           const float32_t * coef_sos, int16_t gain);
bqciirf_df1_handle_t bqciirf_df1_init(void * objmem, int M,
           const float32_t * coef_sos, int16_t gain);
```

```
stereo_bqriir16x16_df1_handle_t stereo_bqriir16x16_df1_init
          (void * objmem, int M,
           const int16_t * coef_sosl, const int16_t * coef_gl, int16_t gainl,
           const int16_t * coef_sosr, const int16_t * coef_gr, int16_t gainr );
stereo_bqriir32x16_df1_handle_t stereo_bqriir32x16_df1_init
          (void * objmem, int M,
           const int16_t * coef_sosl, const int16_t * coef_gl, int16_t gainl,
           const int16_t * coef_sosr, const int16_t * coef_gr, int16_t gainr);
stereo_bqriir32x32_df1_handle_t stereo_bqriir32x32_df1_init
          (void * objmem, int M,
           const int32_t * coef_sosl, const int16_t * coef_gl, int16_t gainl,
           const int32_t * coef_sosr, const int16_t * coef_gr, int16_t gainr);
stereo_bqriirf_df1_handle_t stereo_bqriirf_df1_init
          (void * objmem, int M,
           const float32_t* coef_sosl, int16_t gainl,
           const float32_t* coef_sosr, int16_t gainr );
```

For delayless filters:

```
bqriir16x16_df1_nd_handle_t bqriir16x16_df1_nd_init(void * objmem, int M,
    const int16_t * coef_sos,
    const int16_t * coef_g,
    int16_t         gain);
bqriir16x16_df2_nd_handle_t bqriir16x16_df2_nd_init(void * objmem, int M,
    const int16_t * coef_sos,
    const int16_t * coef_g,
    int16_t         gain);
bqriir32x16_df1_nd_handle_t bqriir32x16_df1_nd_init(void * objmem, int M,
    const int16_t * coef_sos,
    const int16_t * coef_g,
    int16_t         gain);
bqriir32x16_df2_nd_handle_t bqriir32x16_df2_nd_init(void * objmem, int M,
    const int16_t * coef_sos,
    const int16_t * coef_g,
    int16_t         gain);
bqriir32x32_df1_nd_handle_t bqriir32x32_df1_nd_init(void * objmem, int M,
    const int32_t * coef_sos,
    const int16_t * coef_g,
    int16_t         gain);
bqriir32x32_df2_nd_handle_t bqriir32x32_df2_nd_init(void * objmem, int M,
    const int32_t * coef_sos,
    const int16_t * coef_g,
    int16_t         gain);
bqriirf_df1_nd_handle_t bqriirf_df1_nd_init(void * objmem, int M,
    const float32_t * coef_sos, int16_t gain);
bqriirf_df2_nd_handle_t bqriirf_df2_nd_init(void * objmem, int M,
    const float32_t * coef_sos, int16_t gain);
bqriirf_df2t_nd_handle_t bqriirf_df2t_nd_init(void * objmem, int M,
    const float32_t * coef_sos, int16_t gain);
bqciirf_df1_nd_handle_t bqciirf_df1_nd_init(void * objmem, int M,
    const float32_t * coef_sos, int16_t gain);
stereo_bqriir16x16_df1_nd_handle_t stereo_bqriir16x16_df1_nd_init
                                        (void * objmem, int M,
    const int16_t * coef_sosl, const int16_t * coef_gl, int16_t gainl,
    const int16_t * coef_sosr, const int16_t * coef_gr, int16_t gainr);
stereo_bqriir32x16_df1_nd_handle_t stereo_bqriir32x16_df1_nd_init
                                        (void * objmem, int M,
    const int16_t * coef_sosl, const int16_t * coef_gl, int16_t gainl,
    const int16_t * coef_sosr, const int16_t * coef_gr, int16_t gainr);
stereo_bqriir32x32_df1_nd_handle_t stereo_bqriir32x32_df1_nd_init
                                        (void * objmem, int M,
    const int32_t * coef_sosl, const int16_t * coef_gl, int16_t gainl,
    const int32_t * coef_sosr, const int16_t * coef_gr, int16_t gainr);
stereo_bqriirf_df1_nd_handle_t stereo_bqriirf_df1_nd_init
                                        (void * objmem, int M,
    const float32_t* coef_sosl, int16_t gainl,
    const float32_t* coef_sosr, int16_t gainr);
```

Type	Name	Size	Description
Input			
void*	objmem		allocated memory block
int	M		number of bi-quad sections
int32_t, int16_t, float32_t	coef_sos	M*5	filter coefficients stored in blocks of 5 numbers: b0 b1 b2 a1 a2. For fixed-point functions, fixed point format of filter coefficients is Q1.14 for 16x16 and 32x16, or Q1.30 for 32x32.
int32_t, int16_t, float32_t	coef_sosl	M*5	filter coefficients for the left channel (stereo filters only)
int32_t, int16_t, float32_t	coef_sosr	M*5	filter coefficients for the right channel (stereo filters only)
int16_t	coef_g	M	scale factor for each section, Q15 (for fixed-point functions only).
int16_t	coef_gl	M	scale factor for the left channel (stereo filters only)
int16_t	coef_gr	M	scale factor for the right channel (stereo filters only)
int16_t	gain		total gain shift amount, -48..15
int16_t	gainl		total gain shift amount for the left channel (stereo filters only)
int16_t	gainr		total gain shift amount for the right channel (stereo filters only)

Returns: handle to the object

Request algorithmic delay

```
size_t bqriir16x16_df1_groupDelay(bqriir16x16_df1_handle_t _bqriir);
size_t bqriir16x16_df2_groupDelay(bqriir16x16_df2_handle_t _bqriir);
size_t bqriir32x16_df1_groupDelay(bqriir32x16_df1_handle_t _bqriir);
size_t bqriir32x16_df2_groupDelay(bqriir32x16_df2_handle_t _bqriir);
size_t bqriir32x32_df1_groupDelay(bqriir32x32_df1_handle_t _bqriir);
size_t bqriir32x32_df2_groupDelay(bqriir32x32_df2_handle_t _bqriir);
size_t bqriirf_df1_groupDelay    (bqriirf_df1_handle_t  _bqriir);
size_t bqriirf_df2_groupDelay    (bqriirf_df2_handle_t  _bqriir);
size_t bqriirf_df2t_groupDelay   (bqriirf_df2t_handle_t _bqriir);
size_t bqciirf_df1_groupDelay    (bqciirf_df1_handle_t  _bqriir);
size_t stereo_bqriir16x16_df1_groupDelay(stereo_bqriir16x16_df1_handle_t
_bqriir);
size_t stereo_bqriir32x16_df1_groupDelay(stereo_bqriir32x16_df1_handle_t
_bqriir);
size_t stereo_bqriir32x32_df1_groupDelay(stereo_bqriir32x32_df1_handle_t
_bqriir);
size_t stereo_bqriirf_df1_groupDelay    (stereo_bqriirf_df1_handle_t
_bqriir);
```

These functions return additional algorithmic delay introduced by the filter in samples, see more information in para 4.3

For delayless filters:

```
size_t bqriir16x16_df1_nd_groupDelay(bqriir16x16_df1_nd_handle_t _bqriir);
size_t bqriir16x16_df2_nd_groupDelay(bqriir16x16_df2_nd_handle_t _bqriir);
size_t bqriir32x16_df1_nd_groupDelay(bqriir32x16_df1_nd_handle_t _bqriir);
size_t bqriir32x16_df2_nd_groupDelay(bqriir32x16_df2_nd_handle_t _bqriir);
size_t bqriir32x32_df1_nd_groupDelay(bqriir32x32_df1_nd_handle_t _bqriir);
size_t bqriir32x32_df2_nd_groupDelay(bqriir32x32_df2_nd_handle_t _bqriir);
size_t bqriirf_df1_nd_groupDelay(bqriirf_df1_nd_handle_t  _bqriir);
size_t bqriirf_df2_nd_groupDelay(bqriirf_df2_nd_handle_t  _bqriir);
size_t bqriirf_df2t_nd_groupDelay(bqriirf_df2t_nd_handle_t _bqriir);
size_t bqciirf_df1_nd_groupDelay(bqciirf_df1_nd_handle_t  _bqriir);
size_t stereo_bqriir16x16_df1_nd_groupDelay(stereo_bqriir16x16_df1_nd_handle_t
_bqriir);
size_t stereo_bqriir32x16_df1_nd_groupDelay(stereo_bqriir32x16_df1_nd_handle_t
_bqriir);
size_t stereo_bqriir32x32_df1_nd_groupDelay(stereo_bqriir32x32_df1_nd_handle_t
_bqriir);
size_t stereo_bqriirf_df1_nd_groupDelay(stereo_bqriirf_df1_nd_handle_t
_bqriir);
```

For delayless filters these functions return zero

Update the delay line and compute filter output

```
void bqriir16x16_df1(bqriir16x16_df1_handle_t _bqriir,
             void *  s,int16_t *  r,const int16_t *x, int N);
void bqriir16x16_df2(bqriir16x16_df2_handle_t _bqriir,
             void *  s,int16_t *  r,const int16_t *x, int N);
void bqriir32x16_df1(bqriir32x16_df1_handle_t _bqriir,
             void *  s,int32_t *  r,const int32_t *x, int N);
void bqriir32x16_df2(bqriir32x16_df2_handle_t _bqriir,
             void *  s,int32_t *  r,const int32_t *x, int N);
void bqriir32x32_df1(bqriir32x32_df1_handle_t _bqriir,
             void *  s,int32_t *  r,const int32_t *x, int N);
void bqriir32x32_df2(bqriir32x32_df2_handle_t _bqriir,
             void *  s,int32_t *  r,const int32_t *x, int N);
void bqriirf_df1 (bqriirf_df1_handle_t,
                  float32_t    * r, const float32_t * x, int N);
void bqriirf_df2 (bqriirf_df2_handle_t,
                  float32_t    * r, const float32_t * x, int N);
void bqriirf_df2t (bqriirf_df2t_handle_t,
                  float32_t    * r, const float32_t * x, int N);
void bqciirf_df1 (bqciirf_df1_handle_t,
                  complex_float* r, const complex_float * x, int N);
void stereo_bqriir16x16_df1(stereo_bqriir16x16_df1_handle_t _bqriir,
             void *  s,int16_t *  r,const int16_t *x, int N);
void stereo_bqriir32x16_df1(stereo_bqriir32x16_df1_handle_t _bqriir,
             void *  s,int32_t *  r,const int32_t *x, int N);
void stereo_bqriir32x32_df1(stereo_bqriir32x32_df1_handle_t _bqriir,
             void *  s,int32_t *  r,const int32_t *x, int N);
void stereo_bqriirf_df1 (stereo_bqriirf_df1_handle_t,
                  float32_t    * r, const float32_t * x, int N);
```

For delayless filters:

```
void bqriir16x16_df1_nd(bqriir16x16_df1_nd_handle_t _bqriir, void * s, int16_t *
r, const int16_t *x, int N);
void bqriir16x16_df2_nd(bqriir16x16_df2_nd_handle_t _bqriir, void * s, int16_t *
r, const int16_t *x, int N);
void bqriir32x16_df1_nd(bqriir32x16_df1_nd_handle_t _bqriir, void * s, int32_t *
r, const int32_t *x, int N);
void bqriir32x16_df2_nd(bqriir32x16_df2_nd_handle_t _bqriir, void * s, int32_t *
r, const int32_t *x, int N);
void bqriir32x32_df1_nd(bqriir32x32_df1_nd_handle_t _bqriir, void * s, int32_t *
r, const int32_t *x, int N);
void bqriir32x32_df2_nd(bqriir32x32_df2_nd_handle_t _bqriir, void * s, int32_t *
r, const int32_t *x, int N);
void bqriirf_df1_nd(bqriirf_df1_nd_handle_t  _bqriir, float32_t     * r, const
float32_t     * x, int N);
void bqriirf_df2_nd(bqriirf_df2_nd_handle_t  _bqriir, float32_t     * r, const
float32_t     * x, int N);
void bqriirf_df2t_nd(bqriirf_df2t_nd_handle_t _bqriir, float32_t     * r, const
float32_t     * x, int N);
void bqciirf_df1_nd(bqciirf_df1_nd_handle_t  _bqriir, complex_float * r, const
complex_float * x, int N);
void stereo_bqriir16x16_df1_nd(stereo_bqriir16x16_df1_nd_handle_t _bqriir, void
* s, int16_t * r, const int16_t *x, int N);
void stereo_bqriir32x16_df1_nd(stereo_bqriir32x16_df1_nd_handle_t _bqriir, void
* s, int32_t * r, const int32_t *x, int N);
void stereo_bqriir32x32_df1_nd(stereo_bqriir32x32_df1_nd_handle_t _bqriir, void
* s, int32_t * r, const int32_t *x, int N);
void stereo_bqriirf_df1_nd(stereo_bqriirf_df1_nd_handle_t  _bqriir, float32_t
* r, const float32_t     * x, int N);
```

Type	Name	Size	Description
Input			
int16_t, int32_t, float32_t, complex_float	x	N*S	input samples, Q31, Q15 or floating point. Stereo samples go in interleaved order (left, right)
int	N		length of input sample block
	S		1 for mono, 2 for stereo API
Output			
int16_t,	r	N*S	output data, Q31, Q15 or floating point . Stereo samples

int32_t, float32_t, complex_float			go in interleaved order (left, right)
Temporary			
void*	s		scratch memory area (for fixed-point functions only). Minimum number of bytes depends on selected filter structure and precision (see spreadsheet below) If a particular macro returns zero, then the corresponding IIR doesn't require a scratch area and parameter s may hold zero

Returns: none

Functon	Scratch memory, bytes
bqriir16x16_df1	BQRIIR16X16_DF1_SCRATCH_SIZE(N,M)
bqriir16x16_df2	BQRIIR16X16_DF2_SCRATCH_SIZE(N,M)
bqriir32x16_df1	BQRIIR32X16_DF1_SCRATCH_SIZE(N,M)
bqriir32x16_df2	BQRIIR32X16_DF2_SCRATCH_SIZE(N,M)
bqriir32x32_df1	BQRIIR32X32_DF1_SCRATCH_SIZE(N,M)
bqriir32x32_df2	BQRIIR32X32_DF2_SCRATCH_SIZE(N,M)
stereo_bqriir16x16_df1	STEREO_BQRIIR16X16_DF1_SCRATCH_SIZE(N,M)
stereo_bqriir32x16_df1	STEREO_BQRIIR32X16_DF1_SCRATCH_SIZE(N,M)
stereo_bqriir32x32_df1	STEREO_BQRIIR32X32_DF1_SCRATCH_SIZE(N,M)
stereo_bqriirf_df1	STEREO_BQRIIRF_DF1_SCRATCH_SIZE(N,M)
For delayless filters:	
bqriir16x16_df1_nd	BQRIIR16X16_DF1_ND_SCRATCH_SIZE(N,M)
bqriir16x16_df2_nd	BQRIIR16X16_DF2_ND_SCRATCH_SIZE(N,M)
bqriir32x16_df1_nd	BQRIIR32X16_DF1_ND_SCRATCH_SIZE(N,M)
bqriir32x16_df2_nd	BQRIIR32X16_DF2_ND_SCRATCH_SIZE(N,M)
bqriir32x32_df1_nd	BQRIIR32X32_DF1_ND_SCRATCH_SIZE(N,M)
bqriir32x32_df2_nd	BQRIIR32X32_DF2_ND_SCRATCH_SIZE(N,M)
stereo_bqriir16x16_df1_nd	STEREO_BQRIIR16X16_DF1_ND_SCRATCH_SIZE(N,M)
stereo_bqriir32x16_df1_nd	STEREO_BQRIIR32X16_DF1_ND_SCRATCH_SIZE(N,M)
stereo_bqriir32x32_df1_nd	STEREO_BQRIIR32X32_DF1_ND_SCRATCH_SIZE(N,M)
stereo_bqriirf_df1_nd	STEREO_BQRIIRF_DF1_ND_SCRATCH_SIZE(N,M)

Returned value none

Restrictions x,r,s,coef_g,coef_sos must not overlap
N - must be a multiple of 2
s - whenever supplied must be aligned on an 16-bytes boundary

2.2.2 Lattice Block Real IIR

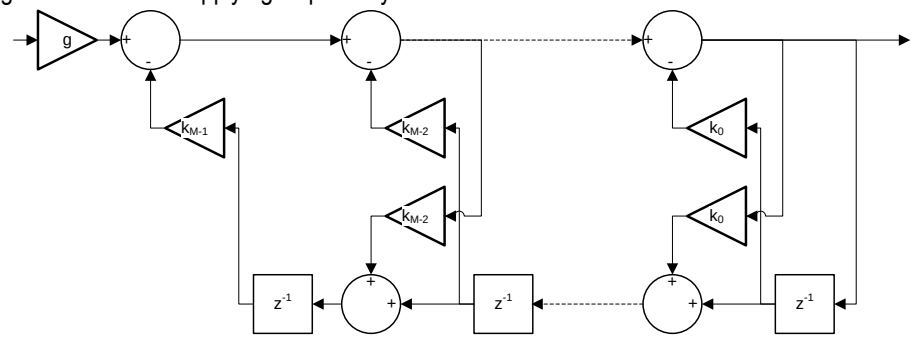
Description Computes a real cascaded lattice autoregressive IIR filter using reflection coefficients stored in vector k. The real data input are stored in vector x. The filter output result is stored in vector r. Input scaling is done before the first cascade for normalization and overflow protection.

Precision 4 variants available:

Type	Description
16x16	16-bit data, 16-bit coefficients
32x16	32-bit data, 16-bit coefficients
32x32	32-bit data, 32-bit coefficients
f	floating point. Requires VFPU/SFPU core option

Algorithm

Algorithm consists of applying sequentially M times IIR sections with structure shown below



Object allocation

```
size_t latr16x16_alloc(int M);
size_t latr32x16_alloc(int M);
size_t latr32x32_alloc(int M);
size_t latrf_alloc    (int M);
```

Type	Name	Size	Description
Input			
int	M		number of sections

Returns: size of memory in bytes to be allocated

Object initialization

```
latr16x16_handle_t latr16x16_init
   (void * objmem, int M,const int16_t    *  k, int16_t   scale);
latr32x16_handle_t latr32x16_init
   (void * objmem, int M, const int16_t   *  k, int16_t   scale);
latr32x32_handle_t latr32x32_init
   (void * objmem, int M, const int32_t   *  k, int32_t   scale);
latrf_handle_t     latrf_init
   (void * objmem, int M, const float32_t *  k, float32_t scale);
```

Type	Name	Size	Description
Input			
void*	objmem		allocated memory block
int	M		number of sections
int16_t, int32_t or float32_t	k	M	reflection coefficients, Q31, Q15 or floating point
int16_t, int32_t or float32_t	scale	M	input scale factor g, Q31, Q15 or floating point

Returns: handle to the object

Update the delay line and compute filter output

```
void latr16x16_process
    (latr16x16_handle_t handle, int16_t   * r, const int16_t   * x, int N);
void latr32x16_process
    (latr32x16_handle_t handle, int32_t   * r, const int32_t   * x, int N);
void latr32x32_process
    (latr32x32_handle_t handle, int32_t   * r, const int32_t   * x, int N);
void latrf_process
    (latrf_handle_t     handle, float32_t * r, const float32_t * x, int N);
```

Type	Name	Size	Description
Input			
`int16_t`, `int32_t` or `float32_t`	`x`	`N`	input samples, Q31, Q15 or floating point
`int`	`N`		length of input sample block
Output			
`int16_t`, `int32_t` or `float32_t`	`r`	`N`	output data, Q31, Q15 or floating point

Returns: none

Returned value none

Restrictions `x,r,k` should not overlap

Conditions for optimum performance For optimum performance `M` should be in range `1…8`

2.2.3 Kalman Filter Update

Description Functions updates Kalman filter gain

Precision 2 variants available:

Type	Description
32x32	32-bit data, 32-bit outputs
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$K = U(H \cdot U + R)^{-1}$$

where

$U = P \cdot H^*$

H - conversion matrix

P - process covariance matrix

Prototype

```
void kalmanupd1_32x32 ( void * pScr,
                        int32_t *K,
                        const int32_t *U,
                        const int32_t *H,
                        const int32_t *R,
                        int N,
                        int qK, int qU, int qH, int qR);
void kalmanupd1f      ( void * pScr,
                        float32_t *K,
                        const float32_t *U,
                        const float32_t *H,
                        const float32_t *R,
                        int N);
```

Scratch allocation functions

```
size_t kalmanupd1_32x32_getScratchSize(int N);
size_t kalmanupd1f_getScratchSize     (int N);
```

Arguments

Type	Name	Size	Description
Parameter			
	M		order - always 1
Input			
int32_t, float32_t	U	[N*M]	state matrix U
int32_t, float32_t	H	[M*N]	measurement matrix
int32_t, float32_t	R	[M*M]	noise estimate (measurement covariance matrix)
int	N		number of states
int	qK		fixed point representation of matrix K, for fixed point API only
int	qU		fixed point representation of matrix U, for fixed point API only
int	qH		fixed point representation of matrix H, for fixed point API only
int	qR		fixed point representation of matrix R, for fixed point API only
Output			
int32_t, float32_t	K	[N*M]	Kalman gain matrix
Temporary			
void	pScr		Scratch memory area of size specified (in bytes) by the corresponding scratch allocation function

Returned value

none

Restrictions

U, R, H, K	should not overlap
U, R, H, K	should be aligned on 16-byte boundary
N	multiple of 32

2.3 Mathematics

A number of DSP Library functions supersede standard floating-point mathematical functions similar to defined in `<math.h>`, as listed below:

ANSI function	Scalar function	reference
atanf	scl_atanf	2.3.10
atan2f	scl_atan2f	2.3.11
cosf	scl_cosinef	2.3.8
sinf	scl_sinef	2.3.8
tanf	scl_tanf	2.3.9
logf	scl_lognf	2.3.3
log2f	scl_log2f	2.3.3
log10f	scl_log10f	2.3.3
expf	scl_antilognf	2.3.4
exp2f	scl_antilog2f	2.3.4
alog10f	scl_antilog10f	2.3.4
tanhf	scl_tanhf	2.3.12

All these functions conform to ISO/IEC 9899 standard (commonly referred to as C99) in respect to function semantics, parameters and return value specification. Moreover, floating-point mathematical functions handle error conditions in a way that differs from general DSP Library approach as stated in 1.8. Aforementioned functions follow the next ground rules:

- Each function executes as if it were a single operation, and may generate any of “invalid”, “overflow” or “divide-by-zero” floating-point exceptions only to reflect the result of that operation.
- A domain error occurs if input argument(s) fall out of the function domain as defined in function specification. In such a case, the function assigns `EDOM` to the integer expression `errno`, raises the “invalid” floating-point exception, and returns a quiet `NaN`.
- `NaN` as an input argument is a special kind of domain error. Namely, the integer expression `errno` acquires `EDOM` and returned value is a quiet `NaN`, but the function raises the “invalid” floating-point exception only if the input argument is a *signaling* `NaN`.
- A floating-point result overflows if the magnitude of the mathematical result is finite but so large that the target floating-point type cannot represent the mathematical result without extraordinary round-off error (for example, `scl_antilognf(100.0f)`). If a function detects a floating-point result overflow, it assigns `ERANGE` to the integer expression `errno`, raises the “overflow” floating-point exception and returns the properly signed infinity value.

The set of floating-point mathematical functions conforming to ISO/IEC 9899 includes vectorized variants of all the functions listed above. Due to the performance reasons, these vectorized functions do not handle `errno` and may generate exceptions in bit different manner to minimize the overhead.

2.3.1 Reciprocal on Q63/Q31/Q15 Numbers

Description

These routines return the fractional and exponential portion of the reciprocal of a vector `x` of Q63, Q31 or Q15 numbers. Since the reciprocal is always greater than 1, it returns fractional portion `frac` in Q(63-`exp`), Q(31-`exp`) or Q(15-`exp`) format and exponent exp so true reciprocal value in the Q0.31/Q0.15 may be found by shifting fractional part left by exponent value.
NOTE: `scl_recip64x64(), scl_recip32x32()` use packed output for mantissa/exponent. To take a full precision, just call vectorized counterparts.

Mantissa accuracy is 1 LSB, so relative accuracy is:

`vec_recip16x16, scl_recip16x16`	6.2e-5
`scl_recip32x32`	2.4e-7
`vec_recip32x32`	9.3e-10
`vec_recip64x64`	2.2e-19

Precision

3 variants available:

Type	Description
64x64	64-bit input, 64-bit output.
32x32	32-bit input, 32-bit output.
16x16	16-bit input, 16-bit output.

Algorithm

$$frac_n \cdot 2^{exp_n} = 1/x_n, n = \overline{0..N-1}$$

Prototype

```
void vec_recip64x64 (int64_t *  frac, int16_t *exp, const int64_t *  x, int N)
void vec_recip32x32 (int32_t *  frac, int16_t *exp, const int32_t *  x, int N)
void vec_recip16x16 (int16_t *  frac, int16_t *exp, const int16_t *  x, int N)
```

Arguments

Type	Name	Size	Description
Input			
`int64_t, int32_t or int16_t`	`x`	`N`	input data, Q63, Q31 or Q15
`int`	`N`		length of vectors
Output			
`int64_t, int32_t or int16_t`	`frac`	`N`	fractional part of result, Q(63-`exp`), Q(31-`exp`) or Q(15-`exp`)
`int16_t`	`exp`	`N`	exponent of result

Returned value

None

Restrictions

`x,frac,exp` should not overlap

Conditions for optimum performance

`frac,x` - aligned on 16-byte boundary
`N` - multiple of 4 and >4

Scalar versions

Prototype

```
uint64_t scl_recip64x64 (int64_t x)
uint32_t scl_recip32x32 (int32_t x)
uint32_t scl_recip16x16 (int16_t x)
```

Arguments

Type	Name	Description
Input		
`int64_t, int32_t or int16_t`	`x`	input data, Q63, Q31 or Q15

Returned value

packed value:
`scl_recip64x64():`
bits 57…0 fractional part
bits 63…58 exponent

`scl_recip32x32():`
bits 23…0 fractional part
bits 31…24 exponent

`scl_recip16x16():`
bits 15…0 fractional part
bits 31…16 exponent

2.3.2 Division of Q63/Q31/Q15 Numbers

Description

These routines perform pair-wise division of vectors written in Q63, Q31 or Q15 format. They return the fractional and exponential portion of the division result. Since the division may generate result greater than 1, it returns fractional portion frac in Q(63-`exp`), Q(31-`exp`) or or Q(15-`exp`) format and exponent exp so true division result in the Q0.63, Q0.31 may be found by shifting fractional part left by exponent value. For division to 0, the result is not defined

Two versions of routines are available: regular versions (`vec_divide32x32, vec_divide16x16`) work with arbitrary arguments, faster versions (`vec_divide32x32_fast, vec_divide16x16_fast`) apply some restrictions.

NOTE: `scl_divide32x32(), scl_divide64x64()` uses packed output for mantissa/exponent. To take a full precision, just call vectorized counterpart.

Mantissa accuracy is 2 LSB, so relative accuracy is:

`vec_divide16x16, scl_divide16x16`	1.2e-4
`scl_divide32x32`	4.8e-7
`vec_divide32x32`	1.8e-9
`vec_divide64x64`	4.3e-19

Precision

4 variants available:

Type	Description
64x64	64-bit inputs, 64-bit output.
64x32i	integer division, 64-bit nominator, 32-bit denominator, 32-bit output
32x32	32-bit inputs, 32-bit output.
16x16	16-bit inputs, 16-bit output.

Algorithm

$$frac_n \cdot 2^{exp_n} = x_n / y_n, n = \overline{0...N-1}$$

Prototype

```
void vec_divide64x64
                (int64_t * frac, int16_t *exp,
                 const int64_t * x, const int64_t * y, int N);
void vec_divide64x32i
                (int32_t * frac, const int64_t * x, const int32_t * y, int N);
void vec_divide32x32
                (int32_t * frac, int16_t *exp,
                 const int32_t *  x, const int32_t *  y, int N)
void vec_divide16x16
                (int16_t *  frac, int16_t *exp,
                 const int16_t *  x, const int16_t *  y, int N)
void vec_divide32x32_fast
                (int32_t *  frac, int16_t *exp,
                 const int32_t *  x, const int32_t *  y, int N);
void vec_divide16x16_fast
                (int16_t *  frac, int16_t *exp,
                 const int16_t *  x, const int16_t *  y, int N);
```

Arguments

Type	Name	Size	Description
Input			

int64_t, int32_t or int16_t	x	N	nominator, 64-bit integer, Q63, Q31 or Q15
int64_t, int32_t or int16_t	y	N	denominator, 32-bit integer, Q63, Q31 or Q15
int	N		length of vectors
Output			
int64_t, int32_t or int16_t	frac	N	fractional parts of result, Q(63-exp), Q(31-exp) or Q(15-exp)
int16_t	exp	N	exponents of result

Returned value

none

Restrictions

For regular versions (vec_divide64x32i, vec_divide64x64, vec_divide32x32, vec_divide16x16) :
x,y,frac,exp should not overlap

For faster versions (vec_divide32x32_fast, vec_divide16x16_fast) :
x,y,frac,exp should not overlap
x, y, frac to be aligned by 16-byte boundary
N - multiple of 4.

Scalar versions

Prototype

```
uint64_t scl_divide64x64(int64_t x,int64_t y);
int32_t  scl_divide64x32(int64_t x,int32_t y);
uint32_t scl_divide32x32 (int32_t x, int32_t y)
uint32_t scl_divide16x16 (int16_t x, int16_t y)
```

Arguments

Type	Name	Description
Input		
int64_t, int32_t or int16_t	x	nominator, 64-bit integer, Q63, Q31 or Q15
int64_t, int32_t or int16_t	y	denominator, 32-bit integer, Q63, Q31 or Q15

Returned value

scl_divide64x64()	packed value: bits 57…0 fractional part,bits 63…58 exponent
scl_divide64x32()	integer remainder
scl_divide32x32()	packed value: bits 23…0 fractional part,bits 31…24 exponent
scl_divide16x16()	packed value: bits 15…0 fractional part, bits 31…16 exponent

2.3.3 Logarithm

Description

Different kinds of logarithm (base 2, natural, base 10). 32-bit fixed point functions interpret input as Q16.15, represent results in Q6.25 format or return 0x80000000 on negative of zero input
Accuracy :

vec_log2_32x32,scl_log2_32x32	730 (2.2e-5)
vec_logn_32x32,scl_logn_32x32	510 (1.5e-5)
vec_log10_32x32,scl_log10_32x32	230 (6.9e-6)
floating point	2 ULP

NOTES:

1. Floating point functions are compatible with standard ANSI C routines and set errno and exception flags accordingly.
2. Floating point functions limit the range of allowable input values:
   - If x<0, the result is set to NaN. In addition, scalar floating point functions assign the value EDOM to errno and raise the "invalid" floating-point exception.
   - If x==0, the result is set to minus infinity. Scalar floating point functions assign the value ERANGE to errno and raise the "divide-by-zero" floating-point exception.

Precision

2 variants available:

Type	Description
32x32	32-bit inputs, 32-bit outputs
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$z_n = \log_K x_n, n = \overline{0...N-1}, K = 2, e, 10$$

Prototypes

```
void vec_log2_32x32 (  int32_t *  z, const int32_t *  x, int N);
void vec_logn_32x32 (  int32_t *  z, const int32_t *  x, int N);
void vec_log10_32x32(  int32_t *  z, const int32_t *  x, int N);
void vec_log2f      (float32_t *  z, const float32_t *  x, int N);
void vec_lognf      (float32_t *  z, const float32_t *  x, int N);
void vec_log10f     (float32_t *  z, const float32_t *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t, float32_t	x	N	input data, Q16.15 or floating point
int	N		length of vectors
Output			
int32_t, float32_t	z	N	Q6.25 or floating point

Returned value

none

Restrictions

`x`, `z` – should not overlap

Scalar versions

Prototypes

```
int32_t scl_log2_32x32 (int32_t x);
int32_t scl_logn_32x32 (int32_t x);
int32_t scl_log10_32x32(int32_t x);
float32_t scl_log2f (float32_t x);
float32_t scl_lognf  (float32_t x);
float32_t scl_log10f(float32_t x);
```

Arguments

Type	Name	Description
Input		
int32_t, float32_t	x	input data, Q16.15 or floating point

Returned value

result, Q6.25 or floating point

2.3.4 Antilogarithm

Description

These routines calculate antilogarithm (base2, natural and base10). 32-bit fixed-point functions accept inputs in Q6.25 and form outputs in Q16.15 format and return 0x7FFFFFFF in case of overflow and 0 in case of underflow.

NOTES:

1. Floating point functions are compatible with standard ANSI C routines and set `errno` and exception flags accordingly.

Precision

2 variants available:

Type	Description
32x32	32-bit inputs, 32-bit outputs. Accuracy: 8e-6*y+1LSB
f	floating point. Accuracy: 2 ULP. Requires VFPU/SFPU core option

Algorithm

$$y_n = 2^{x_n}$$

$$y_n = e^{x_n}$$

$$y_n = 10^{x_n}$$

Prototype

```
void vec_antilog2_32x32(int32_t *  y, const int32_t*  x, int N);
void vec_antilogn_32x32(int32_t *  y, const int32_t*  x, int N);
void vec_antilog10_32x32(int32_t*  y, const int32_t*  x, int N);
void vec_antilog2f (float32_t *  y, const float32_t*  x, int N);
void vec_antilognf (float32_t *  y, const float32_t*  x, int N);
void vec_antilog10f(float32_t *  y, const float32_t*  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t, float32_t	x	N	input data,Q6.25 or floating point
int	N		length of vectors
Output			
int32_t, float32_t	y	N	output data,Q16.15 or floating point

Returned value none

Restrictions x,y – should not overlap

Conditions for optimum performance x,y - aligned on 16-byte boundary
N - multiple of 2

Scalar versions

Prototypes

```
int32_t scl_antilog2_32x32 (int32_t x);
int32_t scl_antilogn_32x32 (int32_t x);
int32_t scl_antilog10_32x32(int32_t x);
float32_t scl_antilog2f (float32_t x);
float32_t scl_antilognf (float32_t x);
float32_t scl_antilog10f(float32_t x);
```

Arguments

Type	Name	Description
Input		
int32_t, float32_t	x	input data, Q6.25 or floating point

Returned value result, Q16.15 or floating point

2.3.5 Power Function

Description

These routines calculates power function for 32-bit fixed-point or floating point numbers. For the fixed point API, The base is represented in Q31, the exponent is represented in Q6.25. Results are represented as normalized fixed point number with separate mantissa in Q31 and exponent..

Notes:
1. Scalar floating point raise to a power function conform to ANSI C requirements on standard math library functions in respect to treatment of errno and floating-point exceptions. Vectorized function does not touch errno and may raise or not raise floating point exceptions.
2. For floating point API, If x<0 is finite, y is finite and not an integer value, then the respective result z is set to NaN, errno is assigned the value EDOM, and the "invalid" floating-point exception is raised.
3. For fixed point API, function returns zero for all non-positive x. Fixed point functions never touch errno

Special cases:

x	y	Result	Extra conditions
floating point functions			
±0	y	±∞	odd y<0
±0	y	+∞	even y<0
±0	y	±0	odd y>0
±0	y	0	even y>0
±1	±∞	1	
1	y	1	any y including NaN

x	±0	1	any x including NaN
x	y	NaN	finite x<0 and finite non-integer y (see note 2 above)
x	−∞	+∞	|x|<1
x	−∞	0	|x|>1
x	+∞	0	|x|<1
x	+∞	+∞	|x|>1
−∞	y	−0	y an odd integer <0
−∞	y	0	y<0 and not an odd integer
−∞	y	−∞	y an odd integer >0
−∞	y	+∞	y>0 and not an odd integer
+∞	y	0	y<0
+∞	y	+∞	y>0
fixed point functions			
x	y	0	x<=0

Precision

2 variants available:

Type	Description
32x32	32-bit inputs, 32-bit outputs. Accuracy: 2 ULP
f	floating point input, floating point output, 2 ULP under condition that |y|<=100

Algorithm

Fixed point API:

$$m_n \cdot 2^{e_n} = x_n^{\,y_n}$$

Floating point API:

$$z_n = x_n^{\,y_n}$$

Prototype

```
void vec_pow_32x32(int32_t *  m, int16_t *e,
                   const uint32_t*  x, const int32_t*  y, int N);
void vec_powf(float32_t *  z,
              const float32_t*  x, const float32_t*  y, int N);
```

Arguments

Type	Name	Size	Description
Input			
uint32_t, float32_t	x	N	input data,Q1.31 (unsigned) or floating point
int32_t	y	N	input data,Q6.25 or floating point
int	N		length of vectors
Output			
int32_t	m	N	mantissa of output, Q31 - for fixed point API
int16_t	e	N	exponent of output - for fixed point API
float32_t	z	N	result: for floating point API

Returned value

none

Restrictions

x,y,z,m – should not overlap

Prototype

```
float32_t scl_powf(float32_t x, float32_t y)
```

Arguments

Type	Name	Size	Description
Input			
float32_t	x		base
float32_t	y		power

Returned value

result

2.3.6 Square Root

Description

These routines calculate square root.
NOTE: functions return 0x80000000 on negative argument for 32-bit outputs or 0x8000 for 16-bit outputs

Two versions of functions available: regular version (vec_sqrt16x16, vec_sqrt32x32, vec_sqrt64x32, vec_sqrt32x16) with arbitrary arguments and faster version

(`vec_sqrt32x32_fast`) that apply some restrictions.

Precision

4 variants available:

Type	Description
16x16	16-bit inputs, 16-bit output. Accuracy: 2 LSB
32x32	32-bit inputs, 32-bit output. Accuracy: (2.6e-7*y+1LSB)
32x16	32-bit input, 16-bit output. Accuracy: 2 LSB
64x32	64-bit input, 32-bit output. Accuracy: 2 LSB

Algorithm

$$y_n = \sqrt{x_n}$$

Prototype

```
void vec_sqrt16x16 (      int16_t*   y, const int16_t  *  x, int N);
void vec_sqrt32x32 (      int32_t*  y, const int32_t*  x, int N);
void vec_sqrt32x16 (      int16_t*  y, const int32_t*  x, int N);
void vec_sqrt64x32 (      int32_t*  y, const int64_t *  x, int N);
void vec_sqrt32x32_fast(  int32_t*  y, const int32_t*  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int64_t`, `int32_t`, `int16_t`	`x`	`N`	input data, Q63, Q31, Q15
`int`	`N`		length of vectors
Output			
`int32_t`, `int16_t`	`y`	`N`	output data, Q31, Q15

Returned value

none

Restrictions

Regular versions (`vec_sqrt16x16, vec_sqrt32x32, vec_sqrt32x16, vec_sqrt64x32`):
`x,y` – should not overlap

Faster versions (`vec_sqrt32x32_fast`):
`x,y` – should not overlap
`x,y` - aligned on 16-byte boundary
`N` - multiple of 2

Scalar versions

Prototypes

```
int16_t  scl_sqrt16x16(int16_t x);
int16_t  scl_sqrt32x16(int32_t x);
int32_t  scl_sqrt32x32(int32_t x);
int32_t  scl_sqrt64x32(int64_t x);
```

Arguments

Type	Name	Description
Input		
`int64_t`, `int32_t`, `int16_t`	`x`	input data, Q63, Q31, Q15

Returned value

result, Q31, Q15

2.3.7 Reciprocal Square Root

Description

These routines return the fractional and exponential portion of the reciprocal square root of a vector `x` of Q31 or Q15 numbers. Since the reciprocal square root is always greater than 1, they return fractional portion `frac` in Q(31-`exp`) or Q(15-`exp`) format and exponent exp so true reciprocal value in the Q0.31/Q0.15 may be found by shifting fractional part left by exponent value.
NOTE: `scl_rsqrt32x32()` uses packed output for mantissa/exponent. To take a full precision, just call vectorized counterpart.
Mantissa accuracy is 1 LSB, so relative accuracy is:

`vec_rsqrt16x16, scl_rsqrt16x16`	6.2e-5
`scl_rsqrt32x32`	2.4e-7
`vec_rsqrt32x32`	9.2e-10

Precision

2 variants available:

Type	Description
32x32	32-bit input, 32-bit output.
16x16	16-bit input, 16-bit output.

Algorithm

$$frac_n \cdot 2^{exp_n} = 1/\sqrt{x_n}, n = \overline{0...N-1}$$

Prototype

```
void vec_rsqrt32x32 (
                 int32_t *  frac, int16_t *exp,
                 const int32_t *  x, int N)
void vec_rsqrt16x16 (
                 int16_t *  frac, int16_t *exp,
                 const int16_t *  x, int N)
```

Arguments

Type	Name	Size	Description
Input			
`int32_t,int16_t`	`x`	`N`	input data, Q31 or Q15
`int`	`N`		length of vectors
Output			
`int32_t, int16_t`	`frac`	`N`	fractional part of result, Q(31-`exp`) or Q(15-`exp`)
`int16_t`	`exp`	`N`	exponent of result

Returned value

None

Restrictions

`x,frac,exp` should not overlap

Scalar versions

Prototype

```
uint32_t scl_rsqrt32x32 (int32_t x)
uint32_t scl_rsqrt16x16 (int16_t x)
```

Arguments

Type	Name	Description
Input		
`int32_t, int16_t`	`x`	input data, Q31 or Q15

Returned value

packed value:
`scl_rsqrt32x32():`
bits 23…0 fractional part
bits 31…24 exponent
`scl_rsqrt16x16():`
bits 15…0 fractional part
bits 31…16 exponent

2.3.8 Sine/Cosine

Description

Fixed-point functions calculate `sin(pi*x)` or `cos(pi*x)` for numbers written in Q31 format. Return results in the same format. Floating point functions compute `sin(x)` or `cos(x)`.
Two versions of functions available: regular version (`vec_sine32x32, vec_cosine32x32, vec_sinef, vec_cosinef`) with arbitrary arguments and faster version (`vec_sine32x32_fast, vec_cosine32x32_fast`) that apply some restrictions.

NOTE:

1. Scalar floating point functions are compatible with standard ANSI C routines and set `errno` and exception flags accordingly.
2. Floating point functions limit the range of allowable input values: [-102940.0, 102940.0] Whenever the input value does not belong to this range, the result is set to `NaN`.

Precision

2 variants available:

Type	Description
32x32	32-bit inputs, 32-bit output. Accuracy: 1700 (7.9e-7)
f	floating point. Accuracy 2 ULP. Requires VFPU/SFPU core option

Algorithm

For fixed point:

$$y_n = \sin(\pi x_n), n = \overline{0...N-1} \text{ or}$$

$$y_n = \cos(\pi x_n), n = \overline{0...N-1}$$

For floating point:

$$y_n = \sin(x_n), n = \overline{0...N-1} \text{ or}$$

$$y_n = \cos(x_n), n = \overline{0...N-1}$$

Prototypes

```
void vec_sine32x32  (  int32_t *  y, const int32_t *  x, int N);
void vec_cosine32x32(int32_t *  y, const int32_t *  x, int N);
void vec_sinef      (  float32_t *  y, const float32_t *  x, int N);
void vec_cosinef    (float32_t *  y, const float32_t *  x, int N);
void vec_sine32x32_fast  (int32_t *  y, const int32_t *  x, int N);
void vec_cosine32x32_fast(int32_t *  y, const int32_t *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t, float32_t	x	N	input data,Q31 or floating point
int	N		length of vectors
Output			
int32_t, float32_t	y	N	Result,Q31 or floating point

Returned value

None

Restrictions

Regular versions (`vec_sine32x32, vec_cosine32x32, vec_sinef, vec_cosinef`):
`x,y` – should not overlap

Faster versions (`vec_sine32x32_fast, vec_cosine32x32_fast`):
`x,y` – should not overlap
`x,y` - aligned on 16-byte boundary
`N` - multiple of 2

Scalar versions

Prototypes

```
int32_t scl_sine32x32   (int32_t x);
int32_t scl_cosine32x32 (int32_t x);
float32_t scl_sinef   (float32_t x);
float32_t scl_cosinef (float32_t x);
```

Arguments

Type	Name	Description
Input		
int32_t, float32_t	x	input data, Q31 or floating point

Returned value

result, Q31 or floating point

2.3.9 Tangent

Description

Fixed point functions calculate `tan(pi*x)` for number written in Q31. Floating point functions compute `tan(x)`.

NOTE:

1. Scalar floating point function is compatible with standard ANSI C routines and sets `errno` and exception flags accordingly.
2. Floating point functions limit the range of allowable input values: [-9099, 9099]. Whenever the input value does not belong to this range, the result is set to `NaN`.

Precision

2 variants available:

Type	Description

32x32	32-bit inputs, 32-bit outputs. Accuracy: (1.3e-4*`y`+1 LSB) if `abs(y)<=464873` (`14.19` in Q15) or `abs(x)<pi*0.4776`
f	floating point, Accuracy: 2 ULP. Requires VFPU/SFPU core option

Algorithm

for fixed point:

$$y_n = \tan(\pi x_n), n = \overline{0...N-1}$$

for floating point

$$y_n = \tan(x_n), n = \overline{0...N-1}$$

Prototype

```
void vec_tan32x32 (int32_t*    y, const int32_t * x, int N);
void vec_tanf (float32_t * y, const float32_t * x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int32_t, float32_t`	`x`	`N`	input data, Q31 or floating point
`int`	`N`		length of vectors
Output			
`int32_t, float32_t`	`y`	`N`	result, Q16.15 or floating point

Returned value

none

Restrictions

`x,y` – should not overlap

Conditions for optimum performance

`x,y` - aligned on 16-byte boundary
`N` - multiple of 2

Scalar versions

Prototype

```
int32_t scl_tan32x32 (int32_t x);
float32_t scl_tanf (float32_t x);
```

Arguments

Type	Name	Description
Input		
`int32_t, float32_t`	`x`	input data,Q31 or floating point

Returned value

result, Q16.15 or floating point

2.3.10 Arctangent

Description

Functions calculate arctangent of number. Fixed point functions scale down the output to `pi`

NOTE:
1. Scalar floating point function is compatible with standard ANSI C routines and sets `errno` and exception flags accordingly

Precision

2 variants available:

Type	Description
32x32	32-bit inputs, 32-bit output. Accuracy: 42 (2.0e-8)
f	floating point. Accuracy: 2 ULP. Requires VFPU/SFPU core option

Algorithm

for fixed point

$$z_n = \arctan(x_n)/\pi, n = \overline{0...N-1}$$

for floating point

$$z_n = \arctan(x_n), n = \overline{0...N-1}$$

Prototype

```
void vec_atan32x32 (int32_t *  z,
              const int32_t *  x,
              int N );
void vec_atanf (    float32_t *  z,
              const float32_t *  x,
              int N );
```

Arguments

Type	Name	Size	Description
Input			
int32_t, float32_t	x	N	input data, Q31 or floating point
int	N		length of vectors
Output			
int32_t, float32_t	z	N	result, Q31 or floating point

Returned value None

Restrictions `x,z` should not overlap

Conditions for optimum performance `x,z` aligned on 16-byte boundary
`N` multiple of 2

Scalar versions

Prototype

```
int32_t scl_atan32x32 (int32_t x);
float32_t scl_atanf (float32_t x);
```

Arguments

Type	Name	Description
Input		
int32_t, float32_t	x	input data, Q31 or floating point

Returned value result, Q31 or floating point

2.3.11 Full Quadrant Arctangent

Description The functions compute the full quadrant arc tangent of the ratio `y/x`. Floating point finctions is in radians. Fixed point functions scale its output by pi.
NOTE:

1. Scalar floating point function is compatible with standard ANSI C routines and sets `errno` and exception flags accordingly
2. Scalar floating point function assigns `EDOM` to `errno` whenever `y==0` and `x==0`.

Precision 1 variant available:

Type	Description
f	floating point. Accuracy: 2 ULP. Requires VFPU/SFPU core option

Algorithm

$$z_n = \arctan(y_n / x_n), n = \overline{0 \ldots N-1}$$

Prototype

```
void vec_atan2f (float32_t * z, const float32_t * y, const float32_t * x,int N);
```

Arguments

Type	Name	Size	Description
Input			
float32_t	x	N	input data, Q31 or floating point
float32_t	y	N	input data, Q31 or floating point
int	N		length of vectors
Output			
float32_t	z	N	result, Q31 or floating point

Returned value None

Restrictions `x, y, z` should not overlap

Scalar versions

Prototype

```
float32_t scl_atan2f (float32_t y, float32_t x);
```

Arguments

Type	Name	Description
Input		
float32_t	y	input data, Q31 or floating point
float32_t	x	input data, Q31 or floating point

Returned value result, Q31 or floating point

2.3.12 Hyperbolic Tangent

Description The functions compute the hyperbolic tangent of input argument. 32-bit fixed-point functions accept inputs in Q6.25 and form outputs in Q16.15 format.

Precision 3 variants available:

Type	Description
32x32	32-bit inputs, 32-bit output. Accuracy: 2 LSB
f	single precision floating point input, floating point output, Accuracy: 2 ULP
fp16	half precision floating point input, half floating point output, Accuracy: 2 ULP

Algorithm

$$y_n = \tanh(x_n), n = \overline{0 \ldots N-1}$$

Prototype

```
void vec_tanh32x32 (int32_t * y, const int32_t * x,int N);
void vec_tanhf     (float32_t * y, const float32_t * x,int N);
void vec_tanh_fp16 (float16_t * y, const float16_t * x,int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t, float32_t, float16_t	x	N	input data, Q6.25 or floating point
int	N		length of vectors
Output			
int32_t, float32_t, float16_t	y	N	result, Q16.15 or floating point

Returned value None

Restrictions `x, y` should not overlap

Scalar versions

Prototype

```
int32_t scl_tanh32x32 (int32_t x);
float32_t scl_tanhf     (float32_t x);
float16_t scl_tanh_fp16     (float16_t x);
```

Arguments

Type	Name	Description
Input		
int32_t, float32_t, float16_t	x	input data, Q6.25 or floating point

Returned value result, Q16.15 or floating point

2.3.13 Sigmoid

Description The functions compute the sigmoid of input argument. 32-bit fixed-point functions accept inputs in Q6.25 and form outputs in Q16.15 format.

Precision 3 variants available:

Type	Description

32x32	32-bit inputs, 32-bit output. Accuracy: 2 LSB
f	floating point input, floating point output. Accuracy 2 ULP
fp16	half precision floating point input, half floating point output, Accuracy: 2 ULP

Algorithm

$$y_n = \frac{1}{1+\exp(-x_n)}, n = \overline{0...N-1}$$

Prototype

```
void vec_sigmoid32x32 (int32_t   * y, const int32_t   * x,int N);
void vec_sigmoidf     (float32_t * y, const float32_t * x,int N);
void vec_sigmoid_fP16 (float16_t * y, const float16_t * x,int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t, float32_t, float16_t	x	N	input data, Q6.25 or floating point
int	N		length of vectors
Output			
int32_t, float32_t, float16_t	y	N	result, Q16.15 or floating point

Returned value

None

Restrictions

`x, y` should not overlap

Scalar versions

Prototype

```
int32_t scl_sigmoid32x32 (int32_t   x);
float32_t scl_sigmoidf     (float32_t x);
float16_t scl_sigmoid_fp16     (float16_t x);
```

Arguments

Type	Name	Description
Input		
int32_t, float32_t, float16_t	x	input data, Q6.25 or floating point

Returned value

result, Q16.15 or floating point

2.3.14 Rectifier function

Description

The functions compute the rectifier linear unit function of input argument. 32-bit fixed-point functions accept inputs in Q6.25 and form outputs in Q16.15 format. Parameter `K` allows to set upper threshold for proper compression of output signal.

Precision

2 variants available:

Type	Description
32x32	32-bit inputs, 32-bit output. Accuracy: 2 LSB
f	floating inputs, floating output. Accuracy: 2 ULP

Algorithm

$$y_n = \max(0, \min(x, K), n = \overline{0...N-1}$$

Prototype

```
void vec_relu32x32 (int32_t   * y, const int32_t   * x, int32_t   K, int N);
void vec_reluf     (float32_t * y, const float32_t * x, float32_t K, int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t, float32_t	x	N	input data, Q6.25 or floating point
int32_t, float32_t	K		threshold, Q16.15 or floating poin
int	N		length of vectors
Output			
int32_t,	y	N	result, Q16.15 or floating poin

`float32_t`			

Returned value None

Restrictions `x, y` should not overlap

Scalar versions

Prototype

```
int32_t   scl_relu32x32 (int32_t   x, int32_t   K);
float32_t scl_reluf     (float32_t x, float32_t K);
```

Arguments

Type	Name	Description
Input		
`int32_t`	`x`	input data, Q6.25 or floating point
`int32_t`	`K`	threshold, Q16.15 or floating point

threshould

Returned value result, Q16.15 or floating point

2.3.15 Softmax

Description The function computes the softmax (normalized exponential function) of input data. 32-bit fixed-point functions accept inputs in Q6.25 and form outputs in Q16.15 format.

Precision 2 variants available:

Type	Description
32x32	32-bit inputs, 32-bit output. Accuracy: 2 LSB (see Note below)
f	floating point input, floating point output

Note: Accuracy of function may depend on amount of data and their distribution. Given accuracy is achieved for N=2 for any pair of data from input domain.

Algorithm

$$y_n = \frac{\exp(x_n)}{\sum_k \exp(x_k)}, n = \overline{0...N-1}$$

Prototype

```
void vec_softmax32x32 (int32_t   * y, const int32_t   * x,int N);
void vec_softmaxf     (float32_t * y, const float32_t * x,int N);
```

Arguments

Type	Name	Size	Description
Input			
`int32_t, float32_t`	`x`	`N`	input data, Q6.25 or floating point
`int`	`N`		length of vectors
Output			
`int32_t, float32_t`	`y`	`N`	result, Q16.15 or floating point

Returned value None

Restrictions `x, y` should not overlap

2.3.16 Integer to Float Conversion

Description Routine converts integer to float and scales result up by `2^t`.

Precision 1 variant available:

Type	Description
f	32-bit input, floating point output. Requires VFPU/SFPU core option

Algorithm

$$y_n = x_n \cdot 2^t, n = \overline{0...N-1}$$

Prototype

```
void   vec_int2float
      ( float32_t  *  y,
```

```
const int32_t  *  x,
int t, int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t	x	N	input data, integer
int	t		scale factor
int	N		length of vectors
Output			
float32_t	y	N	Conversion result, floating point

Returned value None

Restrictions `t` should be in range `-126...126`

Scalar version

Prototype

```
float32_t scl_int2float (int32_t x, int t);
```

Arguments

Type	Name	Description
Input		
int32_t	x	input data, integer

Returned value result, floating point

Restrictions `t` should be in range `-126...126`

2.3.17 Float to Integer Conversion

Description Routine scales floating point input down by `2^t` and converts it to integer with saturation

Precision 1 variant available:

Type	Description
f	floating point input, 32-bit output. Requires VFPU core option

Algorithm

$$y_n = x_n \cdot 2^{-t}, n = \overline{0...N-1}$$

Prototype

```
void    vec_float2int
           (  int32_t *  y,
              const float32_t *  x,
              int t, int N);
```

Arguments

Type	Name	Size	Description
Input			
float32_t	x	N	input data, floating point
int	t		scale factor
int	N		length of vectors
Output			
int32_t	y	N	Conversion results, integers

Returned value None

Restrictions `t` should be in range `-126...126`

Scalar version

Prototype

```
int32_t scl_float2int (float32_t x, int t);
```

Arguments

Type	Name	Description
Input		
float32_t	x	input data, floating point

Returned value result, integer

Restrictions `t` should be in range `-126...126`

2.4 Complex Mathematics

2.4.1 Complex Magnitude

Description Routines compute complex magnitude or its reciprocal

Precision 1 variant available:

Type	Description
f	floating point input, 32-bit output. Requires VFPU/SFPU core option

Algorithm

$$y_n = abs(x_n), n = \overline{0...N-1}$$

$$y_n = 1/abs(x_n), n = \overline{0...N-1}$$

Prototype

```
void vec_complex2mag   (float32_t  *  y, const complex_float  *  x, int N);
void vec_complex2invmag (float32_t  *  y, const complex_float  *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
complex_float	x	N	input data
int	N		length of vectors
Output			
float32_t	y	N	magnitude or its reciprocal

Returned value None

Restrictions None

Scalar version

Prototype

```
float32_t  scl_complex2mag    (complex_float x);
float32_t  scl_complex2invmag (complex_float x);
```

Arguments

Type	Name	Description
Input		
complex_float	x	input data

Returned value result, floating point

Restrictions None

2.5 Vector Operations

2.5.1 Vector Dot Product

Description

These routines take two vectors and calculates their dot product. Two versions of routines are available: regular versions (`vec_dot32x16, vec_dot32x32, vec_dot16x16, vec_dotf`) work with arbitrary arguments, faster versions (`vec_dot32x16_fast, vec_dot32x32_fast, vec_dot16x16_fast`) apply some restrictions.

Precision

7 variants available:

Type	Description
64x32	64x32-bit data, 64-bit output (fractional multiply Q63xQ31->Q63)
64x64	64x64-bit data, 64-bit output (fractional multiply Q63xQ63->Q63)
64x64i	64x64-bit data, 64-bit output (low 64 bit of integer multiply)
32x16	32x16-bit data, 64-bit output
32x32	32x32-bit data, 64-bit output
16x16	16x16-bit data, 64-bit output for regular version and 32-bit for fast version
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$r = \sum_{n=0}^{N-1} x_n y_n$$

Prototype

```
int64_t vec_dot64x32 (const int64_t   *  x,const int32_t * y, int N);
int64_t vec_dot64x64 (const int64_t   *  x,const int64_t * y, int N);
int64_t vec_dot64x64i(const int64_t   *  x,const int64_t * y, int N);
int64_t vec_dot32x16 (const int32_t   *x, const int16_t  * y, int N);
int64_t vec_dot16x16 (const int16_t   *x, const int16_t  * y, int N);
int64_t vec_dot32x32 (const int32_t   *x, const int32_t  * y, int N);
float32_t vec_dotf   (const float32_t *x, const float32_t* y, int N);

int64_t vec_dot64x32_fast (const int64_t * x,const int32_t * y,int N);
int64_t vec_dot64x64_fast (const int64_t * x,const int64_t * y,int N);
int64_t vec_dot64x64i_fast(const int64_t * x,const int64_t * y,int N);
int64_t vec_dot32x16_fast (const int32_t *  x, const int16_t *  y, int N);
int64_t vec_dot32x32_fast (const int32_t *  x, const int32_t *  y, int N);
int32_t vec_dot16x16_fast (const int16_t *  x, const int16_t *  y, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int64_t, int32_t, int16_t, float32_t`	`x`	`N`	input data, Q63, Q31, Q15 or floating point
`int64_t, int16_t, float32_t`	`y`	`N`	input data, Q63, Q31, Q15 or floating point
`int`	`N`		length of vectors

Returned value

dot product of all data pairs, Q63, Q31 or floating point

Restrictions

Regular versions (`vec_dot64x32, vec_dot64x64, vec_dot64x64i, vec_dot32x16, vec_dot32x32,vec_dot16x16, vec_dotf`):
None

Faster versions (`vec_dot64x32_fast, vec_dot64x64_fast, vec_dot64x64i_fast, vec_dot32x16_fast, vec_dot32x32_fast, vec_dot16x16_fast`):
`x,y` - aligned on 16-byte boundary
`N` - multiple of 4

`vec_dot16x16_fast` utilizes 32-bit saturating accumulator, so, input data should be scaled properly to avoid erroneous results especially in case of heterogenic data.

2.5.2 Batch Computation of Vector Dot Products

Description

These routines take a set of input vectors and compute their dot product with specific reference data. Two versions of routines are available:

- regular versions (`vec_dot_batch8x8, vec_dot_batch8x16, vec_dot_batch16x16, vec_dot_batchf, vec_dot_batch_fp16`). They work with arbitrary arguments
- fast versions (`vec_dot_batch8x8_fast, vec_dot_batch8x16_fast, vec_dot_batch16x16_fast, vec_dot_batchf_fast, vec_dot_batch_fp16_fast`) apply some restrictions.

Precision

5 variants available:

Type	Description
8x8	8x8-bit data, 16-bit output (fractional multiply Q7xQ7->Q15)
8x16	8x16-bit data, 16-bit output (fractional multiply Q7xQ15->Q15)
16x16	16x16-bit data, 16-bit output (fractional multiply Q15xQ15->Q31)
f	single precision floating point
fp16	half precision floating point

Algorithm

$$z = 2^{-rsh} \sum_{n=0}^{N-1} x_n y_n^{(m)}$$

Prototype

```
typedef const int8_t   * cint8ptr_t ;
typedef const int16_t   * cint16ptr_t ;
typedef const float32_t * cfloat32ptr_t ;
typedef const float16_t * cfloat16ptr_t ;

void vec_dot_batch8x8  (int16_t *z, const int8_t * x,
                        const cint8ptr_t * y,int rsh, int N, int M);
void vec_dot_batch8x16 (int16_t *z, const int8_t * x,
                        const cint16ptr_t* y,int rsh, int N, int M);
void vec_dot_batch16x16(int32_t *z, const int16_t * x,
                        const cint16ptr_t   * y,int rsh, int N, int M);
void vec_dot_batchf    (float32_t *z, const float32_t * x,
                        const cfloat32ptr_t * y, int N, int M);
void vec_dot_batch8x8_fast  (int16_t *z, const int8_t * x,
                        const cint8ptr_t * y,int rsh, int N, int M);
void vec_dot_batch8x16_fast (int16_t *z, const int8_t * x,
                             const cint16ptr_t   * y,int rsh, int N, int M);
void vec_dot_batch16x16_fast(int32_t *z, const int16_t * x,
                             const cint16ptr_t   * y,int rsh, int N, int M);
void vec_dot_batchf_fast    (float32_t *z, const float32_t * x,
                             const cfloat32ptr_t * y, int N, int M);
void vec_dot_batch_fp16_fast    (float16_t *z, const float16_t * x,
                             const cfloat16ptr_t * y, int N, int M);
```

Arguments

Type	Name	Size	Description
Input			
`int8_t` `int16_t,` `float32_t,` `float16_t`	`x`	`N`	reference vector, Q7, Q15 or floating point
`const int8_t,` `const int16_t,` `const float32_t,` `float16_t`	`y`	`[M][N]`	pointers to M input vectors, Q7, Q15 or floating point
`int`	`rsh`		additional right shift (for fixed point only)
`int`	`N`		length of vectors
`int`	`M`		number of vectors
Output:			
`int16_t,` `int32_t,` `float32_t,` `float16_t`	`z`	`M`	dot products between between reference vector and M input vector, Q15, Q31 or floating point

Returned value none

Restrictions Regular versions: none

Fast versions:

`x, y[m]`	aligned on 16-byte boundary
`N`	multiple of 8
`M`	multiple of 4

2.5.3 Vector Sum

Description This routine makes pair wise saturated summation of vectors. Two versions of routines are available: regular versions (`vec_add32x32, vec_add16x16, vec_addf`) work with arbitrary arguments, faster versions (`vec_add32x32_fast, vec_add16x16_fast`) apply some restrictions.

Precision 3 variants available:

Type	Description
32x32	32-bit inputs, 32-bit output
16x16	16-bit inputs, 16-bit output
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$z_n = x_n + y_n, n = \overline{0...N-1}$$

Prototype

```
void vec_add32x32 ( int32_t*  z, const int32_t*  x, const int32_t*  y, int N);
void vec_add16x16 ( int16_t*  z, const int16_t*  x, const int16_t*  y, int N);
void vec_addf(float32_t*  z, const float32_t*  x, const float32_t*  y, int N);

void vec_add32x32_fast(int32_t*  z, const int32_t*  x, const int32_t*  y,int N);
void vec_add16x16_fast(int16_t*  z, const int16_t*  x, const int16_t*  y,int N);
```

Arguments

Type	Name	Size	Description
Input			
`int32_t, int16_t or float32_t`	`x`	`N`	input data
`int32_t, int16_t or float32_t`	`y`	`N`	input data
`int`	`N`		length of vectors
Output			
`int32_t, int16_t or float32_t`	`z`	`N`	output data

Returned value none

Restrictions Regular versions (`vec_add32x32, vec_add16x16, vec_addf`):
`x,y,z` - should not be overlapped

Faster versions (`vec_add32x32_fast, vec_add16x16_fast`):
`z,x,y` - aligned on 16-byte boundary
`N` - multiple of 4

2.5.4 Power of a Vector

Description These routines compute power of vector with scaling output result by `rsh` bits. Fixed point rountines make accumulation in the 64-bit wide accumulator and output may scaled down with saturation by `rsh` bits. So, if representation of `x` input is `Qx`, result will be represented in `Q(2x-rsh)` format.
Two versions of routines are available: regular versions (`vec_power32x32, vec_power16x16, vec_powerf`) work with arbitrary arguments,
faster versions (`vec_power32x32_fast, vec_power16x16_fast`) apply some restrictions.

Precision

3 variants available:

Type	Description
32x32	32x32-bit data, 64-bit output
16x16	16x16-bit data, 64-bit output
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$r = \frac{1}{2^{rsh}} \sum_{n=0}^{N-1} |x_n|^2$$

Prototype

```
int64_t     vec_power32x32 ( const int32_t *  x,
                             int rsh, int N);
int64_t     vec_power16x16 ( const int16_t *  x,
                             int rsh, int N);
float32_t   vec_powerf     ( const float32_t *  x, int N);

int64_t     vec_power32x32_fast ( const int32_t *  x,
                             int rsh, int N)
int64_t     vec_power16x16_fast ( const int16_t *  x,
                             int rsh, int N)
```

Arguments

Type	Name	Size	Description
Input			
int32_t, int16_t, float32_t	x	N	input data, Q31, Q15 or floating point
int	rsh		right shift of result (only for fixed point routines): for vec_power32x32(): rsh should be in range 31...62 for vec_power16x16(): rsh should be in range 0...31
int	N		length of vector

Returned value

Sum of squares of a vector, Q(2x-rsh) or floating point

Restrictions

For regular versions (vec_power32x32, vec_power16x16, vec_powerf):
none

For faster versions (vec_power32x32_fast, vec_power16x16_fast)
x - aligned on 16-byte boundary
N - multiple of 4

2.5.5 Vector Scaling with Saturation

Description

These routines make shift with saturation of data values in the vector by given scale factor (degree of 2). Functions vec_scale() make multiplication of vector to coefficient which is not a power of 2.
Two versions of routines are available: regular versions (vec_shift32x32, vec_shift16x16, vec_shiftf, vec_scale32x32, vec_scale16x16, vec_scalef, vec_scale_sf) work with arbitrary arguments, faster versions (vec_shift32x32_fast, vec_shift16x16_fast, vec_scale32x32_fast, vec_scale16x16_fast) apply some restrictions.
For floating point:
Function vec_shiftf() makes scaling without saturation of data values in the vector by given scale factor (degree of 2). Functions vec_scalef() and vec_scale_sf() make multiplication of input vector to coefficient which is not a power of 2. vec_scalef() makes scaling without saturations, vec_scale_sf() allows to saturate results on given boundaries.

Precision

3 variants available:

Type	Description
32x32	32-bit input, 32-bit output
16x16	16-bit input, 16-bit output
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$r_n = x_n \cdot 2^t$$

Prototype

```
void vec_shift32x32 (     int32_t *  y,
                    const int32_t *  x, int t, int N);
void vec_shift16x16 (     int16_t *  y,
                    const int16_t *  x, int t, int N);
void vec_shiftf     (     float32_t *  y,
                    const float32 *  x, int t, int N);
void vec_shift32x32_fast (     int32_t *  y,
                    const int32_t *  x, int t, int N);
void vec_shift16x16_fast (     int16_t *  y,
                    const int16_t *  x, int t, int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t, int16_t or float32_t	x	N	input data, Q31, Q15 or floating point
int	t		shift count. If positive, it shifts left with saturation, if negative it shifts right
int	N		length of vector
Output			
int32_t, int16_t or float32_t	y	N	output data, Q31, Q15 or floating point

Prototype

non-power 2 scaling

```
void vec_scale32x32 (     int32_t *  y,
                    const int32_t *  x, int32_t s, int N);
void vec_scale16x16 (     int16_t *  y,
                    const int16_t *  x, int16_t s, int N);
void vec_scalef     (     float32_t *  y,
                    const float32_t *  x, float32_t s, int N);
void vec_scale_sf   (     float32_t * y,
                    const float32_t * x,
                    float32_t s, float32_t fmin, float32_t fmax, int N);
void vec_scale32x32_fast (     int32_t*  y,
                    const int32_t *  x, int32_t s, int N);
void vec_scale16x16_fast (     int16_t *  y,
                    const int16_t *  x, int16_t s, int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t, int16_t or float32_t	x	N	input data, Q31, Q15 or floating point
int16_t, float32_t	s		scale factor, Q31, Q15 or floating point
int	N		length of vector
float32_t	fmin		lower bound of resulted values (for `vec_scale_sf()` only)
float32_t	fmax		upper bound of resulted values (for `vec_scale_sf()` only)
Output			
int32_t, int16_t or float32_t	y	N	output data, Q31, Q15 or floating point

Returned value

None

Restrictions

For regular versions (`vec_shift32x32, vec_shift16x16, vec_shiftf, vec_scale32x32, vec_scale16x16, vec_scalef, vec_scalesf`):
`x,y` should not overlap
`t` should be in range -31...31 for fixed-point functions and -129...146 for floating point

For faster versions (`vec_shift32x32_fast, vec_shift16x16_fast, vec_scale32x32_fast, vec_scale16x16_fast`):
`x,y` should not overlap
`t` should be in range -31...31
`x,y` - aligned on 16-byte boundary

`N` - multiple of 4

2.5.6 Common Exponent

Description

These functions determine the number of redundant sign bits for each value (as if it was loaded in a 32-bit register) and returns the minimum number over the whole vector. This may be useful for a FFT implementation to normalize data.
Floating point function returns `0-floor(log2(max(abs(x))))`. Returned result will be always in range `[-129...146]`.
Special cases

Input	Result
0	0
+/-Inf	-129
NaN	0

NOTES:
Faster versions of functions make the same task but in a different manner – they compute exponent of maximum absolute value in the array. It allows faster computations but not bitexact results – if minimum value in the array will be -2^n, fast function returns `max(0,30-n)` while non-fast function returns `(31-n)`.
Functions return zero if `N<=0`

Precision

3 variants available:

Type	Description
32	32-bit inputs
16	16-bit inputs
f	floating point inputs. Requires VFPU/SFPU core option

Algorithm

$$z_n = \min\left(\underset{n=\overline{0...N-1}}{norm(x_n)} \right) \quad \text{non-fast version}$$

$$z_n = \min\left(\underset{n=\overline{0...N-1}}{norm(abs(x_n))} \right) \quad \text{fast version}$$

$$z_n = -floor\left(\log_2(\max_{n=\overline{0...N-1}}(abs(x_n))) \right) \quad \text{for floating point}$$

where `norm` is exponent value (maximum possible shift count) for 32-bit data.

Prototype

```
int vec_bexp32 (const int32_t *  x, int N);
int vec_bexp16 (const int16_t *  x, int N);
int vec_bexpf  (const float32_t *  x, int N);

int vec_bexp32_fast (const int32_t *  x, int N);
int vec_bexp16_fast (const int16_t *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t, int16_t, float32_t	x	N	input data
int	N		length of vector

Returned value

minimum exponent

Restrictions

non-fast functions (`vec_bexp16, vec_bexp32, vec_bexpf`):
none
for fast functions (`vec_bexp16_fast, vec_bexp32x32_fast`):
`x,y` - aligned on 16-byte boundary
`N` - multiple of 4

Scalar versions

Prototype

```
int scl_bexp32 (int32_t x);
int scl_bexp16 (int16_t x);
int scl_bexpf  (float32_t x);
```

Arguments

Type	Name	Description
Input		
int32_t, int16_t, float32_t	x	input data

Returned value

result

2.5.7 Vector Min/Max

Description

These routines find maximum/minimum value in a vector.
Two versions of functions available: regular version (`vec_min32x32, vec_max32x32, vec_max16x16, vec_min16x16,vec_maxf, vec_minf`) with arbitrary arguments and faster version (`vec_min32x32_fast, vec_max32x32_fast, vec_min16x16_fast, vec_min16x16_fast`) that apply some restrictions
NOTE: functions return zero if `N` is less or equal to zero

Precision

3 variants available:

Type	Description
32x32	32-bit data, 32-bit output
16x16	16-bit data, 16-bit output
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$v = \min(x_n), n = \overline{0...N-1}$$

or

$$v = \max(x_n), n = \overline{0...N-1}$$

Prototype

```
int32_t   vec_min32x32 (const int32_t  *  x, int N);
int16_t   vec_min16x16 (const int16_t  *  x, int N);
float32_t vec_minf     (const float32_t*  x, int N);
int32_t vec_max32x32   (const int32_t  *  x, int N);
int16_t vec_max16x16   (const int16_t  *  x, int N);
float32_t vec_maxf     (const float32_t*  x, int N);
int32_t vec_min32x32_fast (const int32_t*  x, int N);
int16_t vec_min16x16_fast (const int16_t*  x, int N);
int32_t vec_max32x32_fast (const int32_t*  x, int N);
int16_t vec_max16x16_fast (const int16_t*  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t, int16_t, float32_t	x	N	input data
int	N		length of vector

Returned value

minimum or maximum value

Restrictions

For regular routines (`vec_min32x32, vec_max32x32, vec_max16x16, vec_min16x16,vec_maxf, vec_minf`) :
none
For faster routines (`vec_min32x32_fast, vec_max32x32_fast, vec_min16x16_fast, vec_min16x16_fast`):
`x` aligned on 16-byte boundary
`N` - multiple of 4

2.6 Emulated Floating Point Operations

These routines make basic operations with emulated floating point data representing in pairs 32-bit mantissa/16-bit exponent. Mantissa is represented in 2's complement format (as usual integer numbers). Input data might be normalized (positive mantissa from 0x40000000 to 0x7FFFFFFF, negative mantissa from 0x80000000 to 0xBFFFFFFF) or denormalized (mantissa in range from 0xC0000000 to 0x3FFFFFFF). Denormalized numbers are allowed on input, but may cause degraded accuracy. However, on output all functions form normalized numbers.
Exponent bias is 31 so floating point number 1 might be represented as:
- mantissa 0x7FFFFFFF (1 in Q31 representation), exponent 0 (this not exact 1,0 but 0,99999999953)
- mantissa 0x40000000, (1 in Q30 representation), exponent 1
- mantissa 0x20000000, (1 in Q29 representation), exponent 2
and so on.

Special numbers are represented as following:
- zero: mantissa is `0`, exponent does not care. Negative zero is not representable
- positive infinity: mantissa `0x7fffffff`, exponent `0x7fff`
- negative infinity: mantissa `0x80000000`, exponent `0x7fff`
Not a numbers (NaNs) do not have special encoding and cannot be carried by this format.
Comparing with usual single precision floating point computations, these emulation functions

- have wider dynamic range (approximately from 10^{-9873} to 10^{9864})
- have better resolution (31-bit mantissa vs 23-bit in single precision floating point)
- underflowed numbers are converted to zeroes so denormalized numbers are never occurred on the output (but allowable on input)
- result of zero to infinity multiply is not defined
- rounding mode is not specified

It is important to note that similarly as for usual floating point computations, the results of emulated floating point operations may depend on the data order so (a+b)+c may not be exactly the same as a+(b+c) and (a+b)*c may not be equal to a*c+b*c. The difference is explained by different sequence of rounding and normalization.

2.6.1 Vector Addition for Emulated Floating Point

Description add two vectors represented in emulated floating point format

Algorithm

$$z_n = x_n + y_n, n = \overline{0...N-1}$$

Prototype

```
void   vec_add_32x16ef (      int32_t  *  zmant,       int16_t  *  zexp,
                        const int32_t  *  xmant, const int16_t  *  xexp,
                        const int32_t  *  ymant, const int16_t  *  yexp,
                        int N);
```

Arguments

Type	Name	Size	Description
Input			
`int32_t,`	`xmant, ymant`	`N`	mantissa of input data, Q31
`int16_t,`	`xexp, yexp`	`N`	exponent of input data
`int`	`N`		length of vector
Output			
`int32_t,`	`zmant`	`N`	mantissa of output data, Q31
`int16_t,`	`zexp`	`N`	exponent of output data

Returned value none

Scalar function

```
void   scl_add_32x16ef (      int32_t  * zmant, int16_t *  zexp,
                              int32_t    xmant, int16_t    xexp,
                              int32_t    ymant, int16_t    yexp);
```

Arguments

Type	Name	Size	Description
Input			
`int32_t,`	`xmant, ymant`		mantissa of input data, Q31

int16_t,	xexp, yexp		exponent of input data
Output			
int32_t,	zmant	1	mantissa of output data, Q31
int16_t,	zexp	1	exponent of output data

Restrictions `xmant,ymant,xexp,yexp,zmant,zexp` should not overlap

2.6.2 Vector Multiply for Emulated Floating Point

Description multiply two vectors represented in emulated floating point format

Algorithm $z_n = x_n \cdot y_n, n = \overline{0...N-1}$

Prototype

```
void   vec_mul_32x16ef (      int32_t  *  zmant,       int16_t  *  zexp,
                        const int32_t  *  xmant, const int16_t  *  xexp,
                        const int32_t  *  ymant, const int16_t  *  yexp,
                        int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t,	xmant, ymant	N	mantissa of input data, Q31
int16_t,	xexp, yexp	N	exponent of input data
int	N		length of vector
Output			
int32_t,	zmant	N	mantissa of output data, Q31
int16_t,	zexp	N	exponent of output data

Returned value none

Scalar function

```
void   scl_mul_32x16ef (      int32_t  * zmant, int16_t *  zexp,
                              int32_t    xmant, int16_t    xexp,
                              int32_t    ymant, int16_t    yexp);
```

Arguments

Type	Name	Size	Description
Input			
int32_t,	xmant, ymant		mantissa of input data, Q31
int16_t,	xexp, yexp		exponent of input data
Output			
int32_t,	zmant	1	mantissa of output data, Q31
int16_t,	zexp	1	exponent of output data

Restrictions `xmant,ymant,xexp,yexp,zmant,zexp` should not overlap

2.6.3 Vector Multiply-Accumulate for Emulated Floating Point

Description make multiply-accumulate with scalar for data represented in emulated floating point format

Algorithm $z_n = z_n + x_n \cdot y, n = \overline{0...N-1}$

Prototype

```
void   vec_mac_32x16ef (      int32_t  *  zmant,       int16_t  *  zexp,
                        const int32_t  *  xmant, const int16_t  *  xexp,
                              int32_t     ymant,       int16_t     yexp,
                        int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t,	xmant	N	mantissa of input data, Q31
int16_t,	xexp	N	exponent of input data
int32_t,	ymant		mantissa of scalar, Q31
int16_t,	yexp		exponent of scalar

int	N		length of vector
Input/Output			
int32_t,	zmant	N	mantissa of input/output data, Q31
int16_t,	zexp	N	exponent of input/output data

Returned value none

Scalar function

```
void   scl_mac_32x16ef (      int32_t  * zmant, int16_t *  zexp,
                              int32_t    xmant, int16_t    xexp,
                              int32_t    ymant, int16_t    yexp);
```

Arguments

Type	Name	Size	Description
Input			
int32_t,	xmant, ymant		mantissa of input data, Q31
int16_t,	xexp, yexp		exponent of input data
Output			
int32_t,	zmant	1	mantissa of input/output data, Q31
int16_t,	zexp	1	exponent of input/output data

Restrictions `xmant,xexp, zmant,zexp` should not overlap

2.6.4 Vector Dot Product for Emulated Floating Point

Description dot product of input data represented in emulated floating point format

Algorithm

$$z = \sum_{n=0}^{N-1} x_n \cdot y_n$$

Prototype

```
void   vec_dot_32x16ef (      int32_t  *  zmant,       int16_t  *  zexp,
                        const int32_t  *  xmant, const int16_t  *  xexp,
                        const int32_t  *  ymant, const int16_t  *  yexp,
                        int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t,	xmant	N	mantissa of input data, Q31
int16_t,	xexp	N	exponent of input data
int32_t,	ymant	N	mantissa of input data, Q31
int16_t,	yexp	N	exponent of input data
int	N		length of vector
Input/Output			
int32_t,	zmant	1	mantissa of input/output data, Q31
int16_t,	zexp	1	exponent of input/output data

Returned value none

Restrictions `xmant,ymant,xexp,yexp,zmant,zexp` should not overlap

2.7 Matrix Operations

2.7.1 Matrix Multiply

Description

These functions compute the expression `z = 2^lsh * x * y` for the matrices `x` and `y`. The column dimension of `x` must match the row dimension of `y`. The resulting matrix has the same number of rows as `x` and the same number of columns as `y`.
Transposing API allows to interpret input `yt` as transposed (Hermitian transpose for complex data) matrix `y`.
NOTE: `lsh` factor is not relevant for floating point routines.

Functions require scratch memory for storing intermediate data. This scratch memory area should be aligned on 16 byte boundary and its size is calculated by corresponding functions

Two versions of functions available: regular version (`mtx_mpy[t]32x32, mtx_mpy[t]16x16, mtx_mpy[t]8x16, mtx_mpy[t]8x8, mtx[t]_mpyf`) with arbitrary arguments and faster version (`mtx_mpy[t]32x32_fast, mtx_mpy[t]16x16_fast, mtx_mpy[t]8x16_fast, mtx_mpy[t]8x8_fast, mtx_mpy[t]f_fast, cmtx_mpyt32x32_fast, cmtx_mpytf_fast`) that apply some restrictions.

Precision

5 variants available:

Type	Description
32x32	32-bit inputs, 32-bit output
16x16	16-bit inputs, 16-bit output
8x8	8-bit inputs, 8-bit output
8x16	8/16-bit inputs, 16-bit output
f	floating point. Requires VFPU/SFPU core option

Algorithm

For fixed-point routines:

$$z_{m,p} = 2^{lsh}\sum_{n=0}^{N-1} x_{m,n}\cdot y_{n,p}, m = 0..\overline{M-1}, p = \overline{0..P-1}$$

For floating point routines:

$$z_{m,p} = \sum_{n=0}^{N-1} x_{m,n}\cdot y_{n,p}, m = 0..\overline{M-1}, p = \overline{0..P-1}$$

Prototype

```
void mtx_mpy32x32 (  void* pScr,
                     int32_t*  z, const int32_t*  x, const int32_t*  y,
                     int M, int N, int P, int lsh );
void mtx_mpy16x16 (  void* pScr,
                     int16_t*  z, const int16_t*  x, const int16_t*  y,
                     int M, int N, int P, int lsh );
void mtx_mpy8x8  (  void* pScr,
                     int8_t*  z, const int8_t*  x, const int8_t*  y,
                     int M, int N, int P, int lsh );
void mtx_mpy8x16 (  void* pScr,
                     int16_t*  z, const int8_t*  x, const int16_t*  y,
                     int M, int N, int P, int lsh );
void mtx_mpyf (      void* pScr,
                     float32_t*   z, const float32_t*   x, const float32_t*   y,
                     int M, int N, int P);
```

```
void mtx_mpy32x32_fast (  void* pScr,
                  int32_t* z,const int32_t* x, const int32_t* y,
                  int M, int N, int P, int lsh );
void mtx_mpy16x16_fast (  void* pScr,
                  int16_t* z, const int16_t* x, const int16_t* y,
                  int M, int N, int P, int lsh );
void mtx_mpy8x8_fast (  void* pScr,
                  int8_t* z, const int8_t* x, const int8_t* y,
                  int M, int N, int P, int lsh );
void mtx_mpy8x16_fast (  void* pScr,
                  int16_t* z, const int8_t* x, const int16_t* y,
                  int M, int N, int P, int lsh );
void mtx_mpyf_fast (void* pScr,
                  float32_t*   z, const float32_t*   x, const float32_t*   y,
                  int M, int N, int P);
```

Prototype (transposing API)

```
void mtx_mpyt16x16 ( void* pScr,
                     int16_t*  z, const int16_t*  x, const int16_t*  yt,
                     int M, int N, int P, int lsh );
void mtx_mpyt8x8 ( void* pScr,
                     int8_t*  z, const int8_t*  x, const int8_t*  yt,
                     int M, int N, int P, int lsh );
void mtx_mpyt8x16( void* pScr,
                     int16_t*  z, const int8_t*  x, const int16_t*  yt,
                     int M, int N, int P, int lsh );
void mtx_mpyt32x32 ( void* pScr,
                     int32_t*  z, const int32_t*  x, const int32_t*  yt,
                     int M, int N, int P, int lsh );
void mtx_mpytf (     void* pScr,
                     float32_t* z, const float32_t* x, const float32_t* yt,
                     int M, int N, int P);
void mtx_mpyt16x16_fast (  void* pScr,
                     int16_t*  z, const int16_t*  x, const int16_t*  yt,
                     int M, int N, int P, int lsh );
void mtx_mpyt8x8_fast (  void* pScr,
                     int8_t*  z, const int8_t*  x, const int8_t*  yt,
                     int M, int N, int P, int lsh );
void mtx_mpyt8x16_fast (  void* pScr,
                     int16_t*  z, const int8_t*  x, const int16_t*  yt,
                     int M, int N, int P, int lsh );
void mtx_mpyt32x32_fast (  void* pScr,
                     int32_t*  z, const int32_t*  x, const int32_t*  yt,
                     int M, int N, int P, int lsh );
void mtx_mpytf_fast( void* pScr,
                     float32_t* z, const float32_t* x, const float32_t* yt,
                     int M, int N, int P);
void cmtx_mpyt32x32_fast (  void* pScr,
                     complex_fract32*  z,
                     const complex_fract32*  x,
                     const complex_fract32*  yt,
                     int M, int N, int P, int lsh );
void cmtx_mpytf_fast (  void* pScr,
                     complex_float*  z,
                     const complex_float *  x,
                     const complex_float *  yt,
                     int M, int N, int P );
```

Function name	Scratch allocation function
mtx_mpy16x16	mtx_mpy16x16_getScratchSize
mtx_mpy8x8	mtx_mpy8x8_getScratchSize
mtx_mpy8x16	mtx_mpy8x16_getScratchSize
mtx_mpy32x32	mtx_mpy32x32_getScratchSize
mtx_mpyf	mtx_mpyf_getScratchSize
mtx_mpy16x16_fast	mtx_mpy16x16_fast_getScratchSize
mtx_mpy8x8_fast	mtx_mpy8x8_fast_getScratchSize
mtx_mpy8x16_fast	mtx_mpy8x16_fast_getScratchSize
mtx_mpy32x32_fast	mtx_mpy32x32_fast_getScratchSize
mtx_mpyf_fast	mtx_mpyt_fast_getScratchSize
mtx_mpyt16x16	mtx_mpyt16x16_getScratchSize
mtx_mpyt8x8	mtx_mpyt8x8_getScratchSize
mtx_mpyt8x16	mtx_mpyt8x16_getScratchSize
mtx_mpyt32x32	mtx_mpyt32x32_getScratchSize
mtx_mpytf	mtx_mpytf_getScratchSize
mtx_mpyt16x16_fast	mtx_mpyt16x16_fast_getScratchSize
mtx_mpyt32x32_fast	mtx_mpyt32x32_fast_getScratchSize
mtx_mpytf_fast	mtx_mpytf_fast_getScratchSize
cmtx_mpytf_fast	cmtx_mpytf_fast_getScratchSize
cmtx_mpyt32x32_fast	cmtx_mpyt32x32_fast_getScratchSize

Arguments

Type	Name	Size	Description
Input			
int8_t, int16_t, int32_t, float32_t, complex_fract32, complex_float	x	M*N	input matrix,Q31,Q15, Q7 floating point
int8_t, int16_t, int32_t, float32_t, complex_fract32, complex_float	y	N*P	input matrix y. Q31,Q15, Q7 floating point.
int8_t, int16_t, int32_t, float32_t, complex_fract32, complex_float	yt	P*N	transposed (Hermitian transpose for complex data) input matrix y. Q31,Q15, Q7 floating point. (for transposing API only)
int	M		number of rows in matrix x and z
int	N		number of columns in matrix x and number of rows in matrix y
int	P		number of columns in matrices y and z
int	lsh		left shift applied to the result (applied to the fixed-point functions only)
Output			
int8_t, int16_t, int32_t, float32_t, complex_fract32, complex_float	z	M*P	output matrix z,Q31,Q15, Q7 floating point
Temporary			
void*	pScr		Scratch memory area with size in bytes defined by corresponding functions

Returned value

none

Restrictions

For regular routines (mtx_mpy[t]32x32, mtx_mpy[t]16x16, mtx_mpy[t]8x8, mtx_mpy[t]8x16, mtx_mpy[t]f):

x,y,z should not overlap

For faster routines (`mtx_mpy[t]32x32_fast, mtx_mpy[t]16x16_fast, mtx_mpy[t]8x8_fast, mtx_mpy[t]8x16_fast, mtx_mpy[t]f_fast, cmtx_mpyt32x32_fast, cmtx_mpytf_fast`):

`x,y,z`	should not overlap
`x,y,z`	aligned on 16-byte boundary
`M,N,P`	multiplies of 4 for `mtx_mpy[t]32x32_fast, mtx_mpy[t]16x16_fast, mtx_mpy[t]8x8_fast, mtx_mpy[t]8x16_fast, mtx_mpy[t]f_fast` multiplies of 32 for `cmtx_mpyt32x32_fast, cmtx_mpytf_fast`
`lsh`	-31…31 for `mtx_mpy[t]32x32, mtx_mpy[t]32x32_fast, cmtx_mpyt32x32_fast, cmtx_mpytf_fast` -15…15 for `mtx_mpy[t]16x16, mtx_mpy[t]16x16_fast, mtx_mpy[t]8x8, mtx_mpy[t]8x8_fast, mtx_mpy[t]8x16, mtx_mpy[t]8x16_fast`

2.7.2 Matrix by Vector Multiply

Description

These functions compute the expression `z = 2^lsh * x * y` for the matrices `x` and vector `y`.
NOTE: `lsh` factor is not relevant for floating point routines.
Two versions of functions available: regular version (`mtx_vecmpy32x32, mtx_vecmpy16x16, mtx_vecmpy8x8, mtx_vecmpy8x16, mtx_vecmpyf`) with arbitrary arguments and faster version (`mtx_vecmpy32x32_fast, mtx_vecmpy16x16_fast, mtx_vecmpy8x8_fast, mtx_vecmpy8x16_fast, mtx_vecmpyf_fast`) that apply some restrictions.

Precision

5 variants available:

Type	Description
32x32	32-bit inputs, 32-bit output
16x16	16-bit inputs, 16-bit output
8x8	8-bit inputs, 8-bit output
8x16	8/16-bit inputs, 16-bit output
f	floating point. Requires VFPU/SFPU core option

Algorithm

For fixed-point routines:

$$z_n = 2^{lsh} \sum_{m=0}^{M-1} x_{n,m} \cdot y_m, n = 0 \ldots \overline{N-1}$$

For floating-point routines:

$$z_n = \sum_{m=0}^{M-1} x_{n,m} \cdot y_m, n = 0 \ldots \overline{N-1}$$

Prototype

```
void mtx_vecmpy32x32 (  int32_t*  z,
                const int32_t*  x,
                const int32_t*  y,
                int M, int N, int lsh);
void mtx_vecmpy16x16 (  int16_t*  z,
                const int16_t*  x,
                const int16_t*  y,
                int M, int N, int lsh);
void mtx_vecmpy8x8 ( int8_t*  z,
                const int8_t*  x,
                const int8_t*  y,
                int M, int N, int lsh);
void mtx_vecmpy8x16( int16_t*  z,
                const int8_t *  x,
                const int16_t*  y,
                int M, int N, int lsh);
void mtx_vecmpyf (   float32_t*  z,
                const float32_t*  x,
                const float32_t*  y,
                int M, int N);

void mtx_vecmpy32x32_fast (  int32_t*  z,
                const int32_t*  x,
                const int32_t*  y,
                int M, int N, int lsh);
```

```
void mtx_vecmpy16x16_fast (  int16_t*  z,
               const int16_t*  x,
               const int16_t*  y,
               int M, int N, int lsh);
void mtx_vecmpy8x8_fast ( int8_t*  z,
               const int8_t*  x,
               const int8_t*  y,
               int M, int N, int lsh);
void mtx_vecmpy8x16_fast ( int16_t*  z,
               const int8_t *  x,
               const int16_t*  y,
               int M, int N, int lsh);
void mtx_vecmpyf_fast (   float32_t*  z,
               const float32_t*  x,
               const float32_t*  y,
               int M, int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t, int16_t, int8_t, float32_t	x	M*N	input matrix,Q31, Q15, Q7 or floating point
int32_t, int16_t, int8_t, float32_t	y	N	input vector,Q31, Q15, Q7 or floating point
int	M		number of rows in matrix x
int	N		number of columns in matrix x
int	lsh		left shift applied to the result (applied to the fixed-point functions only)
Output			
int32_t, int16_t, int8_t, float32_t	z	M	output vector,Q31, Q15, Q7 or floating point

Returned value

None

Restrictions

For regular routines (mtx_vecmpy32x32, mtx_vecmpy16x16, mtx_vecmpy8x8, mtx_vecmpy8x16, mtx_vecmpyf):

x,y,z should not overlap

For faster routines (mtx_vecmpy32x32_fast, mtx_vecmpy16x16_fast, mtx_vecmpy8x8_fast, mtx_vecmpy8x16_fast, mtx_vecmpyf_fast):

x,y,z should not overlap
x,y,z aligned on 16-byte boundary
N,M multiples of 4
lsh -31...31 for mtx_vecmpy32x32, mtx_vecmpy32x32_fast
-15...15 for mtx_vecmpy16x16, mtx_vecmpy16x16_fast, mtx_vecmpy8x8_fast, mtx_vecmpy8x16_fast

2.7.3 Vector by Vector Multiply

Description

These functions compute the expression y = 2^lsh * x * xt for the input column vector x and its Hermitian transpose xt.
NOTE: lsh factor is not relevant for floating point routines.

Precision

2 variants available:

Type	Description
32x32	32-bit inputs, 32-bit output
f	floating point. Requires VFPU/SFPU core option

Algorithm

For fixed-point routines:

$$y_{m,n} = 2^{lsh} x_m \cdot x_n^*, m = 0 \ldots \overline{N-1}, n = 0 \ldots \overline{N-1}$$

For floating-point routines:

$$y_{m,n} = x_m \cdot x_n^*, m = 0 \ldots \overline{N-1}, n = 0 \ldots \overline{N-1}$$

Prototype

```
void cmtx_vecmpyt32x32_fast (complex_fract32*  y,
                const complex_fract32*  x,
                int N, int lsh);
void cmtx_vecmpytf_fast (complex_float*  y,
                const complex_float *  x,
                int N);
```

Arguments

Type	Name	Size	Description
Input			
complex_fract32, complex_float	x	N	input vector, Q31 or floating point
int	N		size of vector x
int	lsh		bidirectional left shift applied to the result (fixed point functions only).
Output			
complex_fract32, complex_float	y	N*N	output matrix, Q31 or floating point

Returned value

None

Restrictions

x,y	should not overlap
x,y	aligned on 16-byte boundary
N	multiple of 32
lsh	-31...31

2.7.4 Three Matrices Product

Description

These functions compute the expression `z = 2^lsh*((2^-rsh*x*y)*xt)` for square input matrices `x` and `y`, where `xt` is the Hermitian transpose of the matrix `x`.

.

NOTE: `2^lsh` and `2^-rsh` factors are not relevant for floating point routines.

Functions require scratch memory for storing intermediate data. This scratch memory area should be aligned on 16-byte boundary and its size is calculated by corresponding functions.

Precision

2 variants available:

Type	Description
32x32	32-bit inputs, 32-bit output
f	floating point. Requires VFPU/SFPU core option

Algorithm

For fixed-point routines:

$$z = 2^{lsh} \cdot \left(2^{-rsh} \cdot x \cdot y\right) \cdot x^*$$

For floating point routines:

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

Prototype

```
void cmtx_lrmpy32x32_fast (  void* pScr,
                  complex_fract32* z,
                  const complex_fract32* x, const complex_fract32* y,
                  int N, int rsh, int lsh );
void cmtx_lrmpyf_fast (  void* pScr,
                  complex_float* z,
                  const complex_float* x, const complex_float* y,
                  int N );
```

Function name	Scratch allocation function
cmtx_lrmpy32x32_fast	cmtx_lrmpy32x32_fast_getScratchSize
cmtx_lrmpyf_fast	cmtx_lrmpyf_fast_getScratchSize

Arguments

Type	Name	Size	Description
Input			
complex_fract32, complex_float	x	N*N	input matrix x, Q31 or floating point
complex_fract32, complex_float	y	N*N	input matrix y, Q31 or floating point
int	N		number of rows and columns in matrices x, y and z
int	rsh		right shift applied to intermediate results to avoid overflow (fixed point functions only)
int	lsh		bidirectional left shift applied to the result (fixed point functions only)
Output			
complex_fract32, complex_float	z	N*N	output matrix z, Q31 or floating point
Temporary			
void*	pScr		scratch memory area with size in bytes defined by the corresponding scratch allocation function

Returned value

none

Restrictions

x,y,z	should not overlap
x,y,z	aligned on 16-byte boundary
N	multiple of 32
rsh	>=0
lsh	-31...31

2.7.5 Matrix Transpose

Description

These functions transpose matrices.

Precision

4 variants available:

Type	Description
32x32	32-bit inputs, 32-bit output
16x16	16-bit inputs, 16-bit output
8x8	8-bit inputs, 8-bit output
f	floating point

Algorithm

$$y_{n,m} = x_{m,n}, m = 0...\overline{M-1}, n = 0...\overline{N-1}$$

Prototype

```
void mtx_transpose32x32 (int32_t*    y, const int32_t*     x, int M, int N);
void mtx_transpose16x16 (int16_t*    y, const int16_t*     x, int M, int N);
void mtx_transpose8x8   (int8_t *    y, const int8_t *     x, int M, int N);
void mtx_transposef      (float32_t*  y, const float32_t *  x, int M, int N);

void mtx_transpose32x32_fast (int32_t*  y, const int32_t*     x, int M, int N);
void mtx_transpose16x16_fast (int16_t*  y, const int16_t*     x, int M, int N);
void mtx_transpose8x8_fast   (int8_t *  y, const int8_t *     x, int M, int N);
void mtx_transposef_fast    (float32_t*  y, const float32_t *  x, int M, int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t, int16_t, int8_t, float32_t	x	M*N	input matrix,Q31, Q15, Q7 or floating point
int	M		number of rows in matrix x
int	N		number of columns in matrix x
Output			

int32_t, int16_t, int8_t, float32_t	y	M*N	output vector,Q31, Q15, Q7 or floating point

Returned value

None

Restrictions

For regular routines (mtx_transpose_32x32, mtx_transpose_16x16, mtx_transpose_8x8, mtx_transposef):

x,y should not overlap

For faster routines (mtx_transpose 32x32_fast, mtx_transpose 16x16_fast, mtx_transpose_8x8_fast, mtx_transposef_fast):

x,y should not overlap

x,y aligned on 16-byte boundary

N, M multiples of 4

2.8 Matrix Decomposition/Inversion

2.8.1 Gauss-Jordan Matrix Inverse

Description

These functions implement in-place matrix inversion by Gauss elimination with full pivoting.
NOTE: user may detect "invalid" or "divide-by-zero" exception in the CPU flags which MAY indicate that inversion results are not accurate. Also it's responsibility of the user to provide valid input matrix for inversion.
Fixed point version takes representation of input matrix and forms representation of output matrix with proper scaling.

Precision

2 variants available:

Type	Description
f	floating point. Requires VFPU/SFPU core option
32x32	32-bit input, 32-bit output

Algorithm

$y = x^{-1}$

Prototype

Floating point API:

```
void rinvf (void* pScr, float32_t *x);
void cinvf (void* pScr, complex_float *x);
```

Fixed point API:

```
int rinv32x32 (void* pScr, int32_t *x, int qX);
int cinv32x32 (void* pScr, complex_fract32 *x, int qX);
```

N	Function	API	Scratch allocation function
2	mtx_inv2x2f	rinvf	mtx_inv2x2f_getScratchSize
3	mtx_inv3x3f	rinvf	mtx_inv3x3f_getScratchSize
4	mtx_inv4x4f	rinvf	mtx_inv4x4f_getScratchSize
6	mtx_inv6x6f	rinvf	mtx_inv6x6f_getScratchSize
8	mtx_inv8x8f	rinvf	mtx_inv8x8f_getScratchSize
10	mtx_inv10x10f	rinvf	mtx_inv10x10f_getScratchSize
2	cmtx_inv2x2f	cinvf	cmtx_inv2x2f_getScratchSize
3	cmtx_inv3x3f	cinvf	cmtx_inv3x3f_getScratchSize
4	cmtx_inv4x4f	cinvf	cmtx_inv4x4f_getScratchSize
6	cmtx_inv6x6f	cinvf	cmtx_inv6x6f_getScratchSize
8	cmtx_inv8x8f	cinvf	cmtx_inv8x8f_getScratchSize
10	cmtx_inv10x10f	cinvf	cmtx_inv10x10f_getScratchSize
2	mtx_inv2x2_32x32	rinv32x32	mtx_inv2x2_32x32_getScratchSize
3	mtx_inv3x3_32x32	rinv32x32	mtx_inv3x3_32x32_getScratchSize
4	mtx_inv4x4_32x32	rinv32x32	mtx_inv4x4_32x32_getScratchSize
6	mtx_inv6x6_32x32	rinv32x32	mtx_inv6x6_32x32_getScratchSize
8	mtx_inv8x8_32x32	rinv32x32	mtx_inv8x8_32x32_getScratchSize
10	mtx_inv10x10_32x32	rinv32x32	mtx_inv10x10_32x32_getScratchSize
2	cmtx_inv2x2_32x32	cinv32x32	cmtx_inv2x2_32x32_getScratchSize
3	cmtx_inv3x3_32x32	cinv32x32	cmtx_inv3x3_32x32_getScratchSize
4	cmtx_inv4x4_32x32	cinv32x32	cmtx_inv4x4_32x32_getScratchSize
6	cmtx_inv6x6_32x32	cinv32x32	cmtx_inv6x6_32x32_getScratchSize
8	cmtx_inv8x8_32x32	cinv32x32	cmtx_inv8x8_32x32_getScratchSize
10	cmtx_inv10x10_32x32	cinv32x32	cmtx_inv10x10_32x32_getScratchSize

Copyright 2009-2020, IntegrIT Ltd., www.integrit.com

Arguments

Type	Name	Size	Description
Input			
`float32_t, complex_float, int32_t, complex_fract32`	`x`	`N*N`	input matrix
`int`	`qX`		input matrix representation (for fixed point API only)
Output			
`float32_t, complex_float, int32_t, complex_fract32`	`x`	`N*N`	output inverted matrix
Temporary			
`void`	`pScr`		scratch memory. Size in bytes is defined by corresponding scratch allocation function

Returned value

floating functions return none, fixed point functions return fixed-point representation of inverted matrix

Restrictions

none

2.8.2 Gauss-Jordan Matrix Equation Solver

Description

These functions implement solution of matrix equation by Gauss elimination with full pivoting.
Fixed point functions take representation of input matrix and vector and form representation of output vector with proper scaling.

Precision

1 variant available:

Type	Description
32x32	32-bit input, 32-bit output
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$y = Ax$$

Prototype

Floating point API:

```
void rinvf (void* pScr,
               float32_t *x,
               const float32_t *A, const float32_t *x);
void cinvf (void* pScr,
               complex_float *x,
               const complex_float *A, const complex_float *x);
```

Fixed point API:

```
int r32x32 (void* pScr,
               int32_t *y,
               const int32_t *A, const int32_t *x,
               int qA, int qX);
int c32x32 (void* pScr,
               complex_fract32 *x,
               const complex_fract32 *A, const complex_fract32 *x,
               int qA, int qX);
```

N	Function	API	Scratch allocation function
2	mtx_gjelim2x2f	rinvf	mtx_gjelim2x2f_getScratchSize
3	mtx_gjelim3x3f	rinvf	mtx_gjelim3x3f_getScratchSize
4	mtx_gjelim 4x4f	rinvf	mtx_gjelim4x4f_getScratchSize
6	mtx_gjelim 6x6f	rinvf	mtx_gjelim6x6f_getScratchSize
8	mtx_gjelim 8x8f	rinvf	mtx_gjelim8x8f_getScratchSize
10	mtx_gjelim10x10f	rinvf	mtx_gjelim10x10f_getScratchSize
2	mtx_gjelim2x2_32x32	r32x32	mtx_gjelim2x2_32x32_getScratchSize
3	mtx_gjelim3x3_32x32	r32x32	mtx_gjelim3x3_32x32_getScratchSize
4	mtx_gjelim 4x4_32x32	r32x32	mtx_gjelim4x4_32x32_getScratchSize
6	mtx_gjelim 6x6_32x32	r32x32	mtx_gjelim6x6_32x32_getScratchSize
8	mtx_gjelim 8x8_32x32	r32x32	mtx_gjelim8x8_32x32_getScratchSize
10	mtx_gjelim10x10_32x32	r32x32	mtx_gjelim10x10_32x32_getScratchSize
2	cmtx_gjelim 2x2f	cinvf	cmtx_gjelim2x2f_getScratchSize
3	cmtx_gjelim 3x3f	cinvf	cmtx_gjelim3x3f_getScratchSize
4	cmtx_gjelim 4x4f	cinvf	cmtx_gjelim4x4f_getScratchSize
6	cmtx_gjelim 6x6f	cinvf	cmtx_gjelim6x6f_getScratchSize
8	cmtx_gjelim 8x8f	cinvf	cmtx_gjelim8x8f_getScratchSize
10	cmtx_gjelim10x10f	cinvf	cmtx_gjelim10x10f_getScratchSize
2	cmtx_gjelim 2x2_32x32	c32x32	cmtx_gjelim2x2_32x32_getScratchSize
3	cmtx_gjelim 3x3_32x32	c32x32	cmtx_gjelim3x3_32x32_getScratchSize
4	cmtx_gjelim 4x4_32x32	c32x32	cmtx_gjelim4x4_32x32_getScratchSize
6	cmtx_gjelim 6x6_32x32	c32x32	cmtx_gjelim6x6_32x32_getScratchSize
8	cmtx_gjelim 8x8_32x32	c32x32	cmtx_gjelim8x8_32x32_getScratchSize
10	cmtx_gjelim10x10_32x32	c32x32	cmtx_gjelim10x10_32x32_getScratchSize

Copyright 2009-2020, IntegrIT Ltd., www.integrit.com

Arguments

Type	Name	Size	Description
Input			
`float32_t, complex_float, int32_t, complex_fract32`	`A`	`N*N`	input matrix. For fixed point API, representation is defined by parameter qA
`float32_t, complex_float, int32_t, complex_fract32`	`x`	`N`	input rigth side of equation. For fixed point API,, representation is defined by parameter qX
`int`	`qA`		input matrix representation (for fixed point API only)
`int`	`qX`		input vector representation (for fixed point API only)
Output			
`complex_float, int32_t, complex_fract32`	`y`	`N`	output vector
Temporary			
`void`	`pScr`		scratch memory. Size in bytes is defined by corresponding scratch allocation function

Returned value

fixed point functions return fixed-point representation of output vector

Restrictions

none

2.8.3 Cholesky MMSE Solver

Description

Compute the MMSE solution for a system of linear equations `A*x=B`, where `A` is an `MxN` real (complex) matrix with `M>=N` and `rank(A)==N`, `x` is an `Nx1` vector of unknowns, and `B` is an `Mx1` right hand side vector. This task is accomplished in 3 steps:

- Cholesky decomposition is applied to the matrix of normal equations system, which results in an upper triangular matrix R_{NxN} with real and positive numbers on the main diagonal, such that $R^*_{NxN} \cdot R_{NxN} \cong A^*_{MxN} \cdot A_{MxN} + \sigma^2 I_{NxN}$
- Forward substitution step: solve $R^*_{NxN} \cdot y_{NxP} = A^*_{MxN} B_{MxP}$ for `Nx1` vector `y`.
- Backward substitution step: solve the system $R_{NxN} \cdot x_{NxP} = y_{NxP}$ for the `Nx1` vector of unknowns `x`.

For a single `MxN` matrix `A`, these 3 steps may be done simultaneously for `P` variants of `Mx1` right hand side column vectors b gathered into an `MxP` input matrix `B`. MMSE solution is computed independently for each of `P` columns, with resulting column vectors forming the solution matrix `x` of size `NxP`.
Fixed point API requires explicit setting of fixed point representation of input/output matrices as well as for internal temporary matrices such as `R` (Cholesky decomposition) and `y`.

Precision

2 variants available:

Type	Description
f	floating point input, floating point output
32x32	32-bit fixed point

Algorithm

find $R_{NxN} = chol(A^*_{MxN} \cdot A_{MxN} + \sigma^2 I_{NxN})$: $R^*_{NxN} \cdot R_{NxN} \cong A^*_{MxN} \cdot A_{MxN} + \sigma^2 I_{NxN}$

solve $R^*_{NxN} \cdot y_{NxP} = A^*_{MxN} B_{MxP}$

solve $R_{NxN} \cdot x_{NxP} = y_{NxP}$

Prototype, floating point

```
void  rmmse   (void * pScr,
                     float32_t * x,
                 const float32_t * A,
                 const float32_t * B,
                     float32_t sigma2);
void  cmmse    (void * pScr,
                     complex_float * x,
                 const complex_float * A,
```

```
                    const complex_float * B,
                          float32_t sigma2);
```

N, M	Function	API	Scratch allocation function
	Real data		
4	matcholmmsesolver4x4f	rmmse	matcholmmsesolver4x4f_getScratchSize
6	matcholmmsesolver6x6f	rmmse	matcholmmsesolver6x6f_getScratchSize
8	matcholmmsesolver8x8f	rmmse	matcholmmsesolver8x8f_getScratchSize
10	matcholmmsesolver10x10f	rmmse	matcholmmsesolver10x10f_getScratchSize
	Complex data		
4	cmatcholmmsesolver4x4f	cmmse	cmatcholmmsesolver4x4f_getScratchSize
6	cmatcholmmsesolver6x6f	cmmse	cmatcholmmsesolver6x6f_getScratchSize
8	cmatcholmmsesolver8x8f	cmmse	cmatcholmmsesolver8x8f_getScratchSize
10	cmatcholmmsesolver10x10f	cmmse	cmatcholmmsesolver10x10f_getScratchSize

Prototype, fixed point

```
void  rmmse32   (void * pScr,
                       int32_t * x,
                 const int32_t * A,
                 const int32_t * B,
                       int32_t sigma2, int qRA, int qYBRA, int qXYR);
void  cmmse32    (void * pScr,
                       complex_fract32 * x,
                 const complex_ fract32 * A,
                 const complex_ fract32 * B,
                       int32_t  sigma2, int qRA, int qYBRA, int qXYR);
```

N, M	Function	API	Scratch allocation function
	Real 32-bit data		
4	matcholmmsesolver4x4_32x32	rmmse32	matcholmmsesolver4x4_32x32_getScratchSize
6	matcholmmsesolver6x6_32x32	rmmse32	matcholmmsesolver6x6_32x32_getScratchSize
8	matcholmmsesolver8x8_32x32	rmmse32	matcholmmsesolver8x8_32x32_getScratchSize
10	matcholmmsesolver10x10_32x32	rmmse32	matcholmmsesolver10x10_32x32_getScratchSize
	Complex 32-bit data		
4	cmatcholmmsesolver4x4_32x32	cmmse32	cmatcholmmsesolver4x4_32x32_getScratchSize
6	cmatcholmmsesolver6x6_32x32	cmmse32	cmatcholmmsesolver6x6_32x32_getScratchSize
8	cmatcholmmsesolver8x8_32x32	cmmse32	cmatcholmmsesolver8x8_32x32_getScratchSize
10	cmatcholmmsesolver10x10_32x32	cmmse32	cmatcholmmsesolver10x10_32x32_getScratchSize

Arguments

Type	Name	Size	Description
Parameters			
int	M		Dimensional parameter: number of rows in input matrix A
int	N		Dimensional parameter: number of columns in input matrix A
int	P		Number of columns in right-side matrix B
Input:			
float32_t, int32_t	sigma2		Regularization term. . For fixed point, the representation should be Q(2*qA-30)
float32_t, complex_float, int32_t,	A	[M*N]	matrix A . For fixed point API, the representation is Q(qA)

complex_fract32			
float32_t, complex_float, int32_t, complex_fract32	B	[M*P]	Original right-side matrix B. For fixed point, the representation should be Q(qB)
int	qRA		qR-qA (for the fixed point API only); difference between fixed point representations of R and A matrices (for the fixed point API only). Should be equal or less than 0 (typically -2)
int	qYBRA		qY-qB+qR-qA, combination of fixed point representations of matrices y, B, R and A (for the fixed point API only)
int	qXYR		combination of fixed point representation (matrices R, x and y) qXYR=qX-qY+qR
Output			
float32_t, complex_float, int32_t, complex_fract32	x	[N*P]	Decision matrix x. For fixed point, the representation is Q(qX)
Temporary			
	pScr		Scratch data

Returned value

none

Restrictions

```
N = M = 4, 6, 8, 10
P = 1
```

All matrices should not overlap and be aligned on 8-bytes boundary

2.8.4 Cholesky Decomposition for Pseudo-inversion

Description

Apply the Cholesky decomposition to the matrix of normal equations system associated with a complex-valued least squares problem: $A_{MxN} \cdot X_{NxP} = B_{MxP}$, $M \geq N$.
The decomposition results in an upper triangular complex matrix R_{NxN} with real and positive numbers on the main diagonal, such that $R^* \cdot R = A^* \cdot A + \sigma^2 \cdot I_N$, where <…>* denotes the conjugate transpose of a matrix, and $\sigma^2 \cdot I_N$ is the NxN identity matrix multiplied with the regularization term.

Precision

2 variants available:

Type	Description
f	floating point input, floating point output
32x32	32-bit fixed point

Algorithm

$$R_{NxN} = chol(A^*_{MxN} \cdot A_{MxN} + \sigma^2 I_{NxN}):\ R^*_{NxN} \cdot R_{NxN} \cong A^*_{MxN} \cdot A_{MxN} + \sigma^2 I_{NxN}$$

Prototype, floating point

```
void  cchol (void* pScr,
                complex_float *R,
                complex_float *D,
          const complex_float *A,
                float32_t  sigma2);

void  rchol(void* pScr,
                 float32_t *R,
                 float32_t * D,
           const float32_t * A,
                 float32_t  sigma2);
```

N, M	Function	API	Scratch allocation function
	Real data		
4	matcholdecomp4x4f	rchol	matcholdecomp4x4f_getScratchSize
6	matcholdecomp6x6f	rchol	matcholdecomp6x6f_getScratchSize
8	matcholdecomp8x8f	rchol	matcholdecomp8x8f_getScratchSize
10	matcholdecomp10x10f	rchol	matcholdecomp10x10f_getScratchSize
	Complex data		
4	cmatcholdecomp4x4f	cchol	cmatcholdecomp4x4f_getScratchSize

6	cmatcholdecomp6x6f	cchol	cmatcholdecomp6x6f_getScratchSize
8	cmatcholdecomp8x8f	cchol	cmatcholdecomp8x8f_getScratchSize
10	cmatcholdecomp10x10f	cchol	cmatcholdecomp10x10f_getScratchSize

Prototype, floating point

```
void  cchol32 (void* pScr,
               complex_fract32 *R,
               complex_fract32 *D,
         const complex_fract32 *A,
               int32_t  sigma2, int qRA);

void  rchol32(void* pScr,
                int32_t *R,
                complex_fract32 * D,
          const int32_t * A,
                int32_t  sigma2, int qRA);
```

N, M	Function	API	Scratch allocation function
	Real 32-bit data		
4	matcholdecomp4x4_32x32	rchol32	matcholdecomp4x4_32x32_getScratchSize
6	matcholdecomp6x6_32x32	rchol32	matcholdecomp6x6_32x32_getScratchSize
8	matcholdecomp8x8_32x32	rchol32	matcholdecomp8x8_32x32_getScratchSize
10	matcholdecomp10x10_32x32	rchol32	matcholdecomp10x10_32x32_getScratchSize
	Complex 32-bit data		
4	cmatcholdecomp4x4_32x32	cchol32	cmatcholdecomp4x4_32x32_getScratchSize
6	cmatcholdecomp6x6_32x32	cchol32	cmatcholdecomp6x6_32x32_getScratchSize
8	cmatcholdecomp8x8_32x32	cchol32	cmatcholdecomp8x8_32x32_getScratchSize
10	cmatcholdecomp10x10_32x32	cchol32	cmatcholdecomp10x10_32x32_getScratchSize

Arguments

Type	Name	Size	Description
Parameters			
int	M		Dimensional parameter: number of rows in input matrix A
int	N		Dimensional parameter: number of columns in input matrix A, number of rows and columns in upper triangular matrix R
Input			
float32_t, complex_float, int32_t, complex_fract32	A	[M*N]	matrix A . For fixed point API, the representation is Q(qA)
float32_t, int32_t	sigma2		Regularization term. . For fixed point, the representation should be Q(2*qA-30)
int	qRA		qR-qA (for the fixed point API only); difference between fixed point representations of R and A matrices (for the fixed point API only). Should be equal or less than 0 (typically -2)
Output			
float32_t, complex_float, int32_t, complex_fract32	R	((N+1)*N)/2	upper triangular matrix R . For fixed point API, the representation is Q(qR)
float32_t, complex_float,	D	[N]	reciprocals of the main diagonal NOTE: for the fixed point API, these data are

complex_fract32			stored internally in special format with separate mantissa and exponent for better accuracy and dynamic range control. So, even for the real data, they stored as pairs of 2 integers and packed to the complex_fract32 format
Temporary			
	pScr		Scratch memory

Returned value

none

Restrictions

N = M = 4, 6, 8, 10

All matrices and scratch memory should not overlap and be aligned on 8-bytes boundary

2.8.5 Cholesky Forward Substitution for Pseudo-inversion

Description

These functions make forward recursion stage of pseudo-inversion. They use Cholesky decomposition R_{NxN} of original matrices A_{MxN}.

Precision

2 variants available:

Type	Description
f	floating point input, floating point output
32x32	32-bit fixed point

Algorithm

Solution of equations

$$R^*_{NxN} \cdot y_{NxP} = A^*_{MxN} B_{MxP}$$

Prototype, floating point

```
void  cfwd (void * pScr,
                  complex_float * y,
            const complex_float * R,
            const complex_float * D,
            const complex_float * A,
            const complex_float * B);

void  rfwd (void * pScr,
                  float32_t * y,
             const float32_t * R,
             const float32_t * D,
             const float32_t * A,
             const float32_t * B);
```

N, M	Function	API	Scratch allocation function
	Real data		
4	matcholfwdsubst4x4f	rfwd	matcholfwdsubst4x4f_getScratchSize
6	matcholfwdsubst6x6f	rfwd	matcholfwdsubst6x6f_getScratchSize
8	matcholfwdsubst8x8f	rfwd	matcholfwdsubst8x8f_getScratchSize
10	matcholfwdsubst10x10f	rfwd	matcholfwdsubst10x10f_getScratchSize
	Complex data		
4	cmatcholfwdsubst4x4f	cfwd	cmatcholfwdsubst4x4f_getScratchSize
6	cmatcholfwdsubst6x6f	cfwd	cmatcholfwdsubst6x6f_getScratchSize
8	cmatcholfwdsubst8x8f	cfwd	cmatcholfwdsubst8x8f_getScratchSize
10	cmatcholfwdsubst10x10f	cfwd	cmatcholfwdsubst10x10f_getScratchSize

Prototype, fixed point

```
void  cfwd32 (void * pScr,
                  complex_fract32 * y,
            const complex_fract32 * R,
            const complex_fract32 * D,
            const complex_fract32 * A,
            const complex_fract32 * B,
            int qYBRA);
```

```
void  rfwd32 (void * pScr,
                     int32_t * y,
               const int32_t * R,
               const complex_fract32 * D,
               const int32_t * A,
               const int32_t * B,
               int qYBRA);
```

N, M	Function	API	Scratch allocation function
	Real 32-bit data		
4	matcholfwdsubst4x4_32x32	rfwd32	matcholfwdsubst4x4_32x32_getScratchSize
6	matcholfwdsubst6x6_32x32	rfwd32	matcholfwdsubst6x6_32x32_getScratchSize
8	matcholfwdsubst8x8_32x32	rfwd32	matcholfwdsubst8x8_32x32_getScratchSize
10	matcholfwdsubst10x10_32x32	rfwd32	matcholfwdsubst10x10_32x32_getScratchSize
	Complex 32-bit data		
4	cmatcholfwdsubst4x4_32x32	cfwd32	cmatcholfwdsubst4x4_32x32_getScratchSize
6	cmatcholfwdsubst6x6_32x32	cfwd32	cmatcholfwdsubst6x6_32x32_getScratchSize
8	cmatcholfwdsubst8x8_32x32	cfwd32	cmatcholfwdsubst8x8_32x32_getScratchSize
10	cmatcholfwdsubst10x10_32x32	cfwd32	cmatcholfwdsubst10x10_32x32_getScratchSize

Arguments

Type	Name	Size	Description
Parameters			
int	M		Matrix dimension (number of rows in matrix A)
int	N		Matrix dimension (number of columns in matrix A)
int	P		Number of columns in right-side matrix B
Input			
float32_t, complex_float, int32_t, complex_fract32	R	[((N+1)*N)/2]	upper triangular matrix R. For fixed point, representation is Q(qR)
float32_t, complex_float, int32_t, complex_fract32	A	[M*N]	matrix A. For fixed point, representation is Q(qA)
float32_t, complex_float, int32_t, complex_fract32	B	[M*P]	original right-side matrix B. For fixed point, representation is Q(qB)
float32_t, complex_float, complex_fract32	D	[N]	reciprocals of main diagonal. NOTE: for the fixed point API, these data are stored internally in special format with separate mantissa and exponent for better accuracy and dynamic range control. So, even for the real data, they stored as pairs of 2 integers and packed to the complex_fract32 format
int	qYBRA		qY-qB+qR-qA, combination of fixed poin representations of matrices y,B, R and A (for the fixed point API only)
Output			

float32_t, complex_float, int32_t, complex_fract32	y	[N*P]	Decision matrix y. For fixed point, representation is Q(qY)
Temporary			
	pScr		Scratch memory

Returned value

none

Restrictions

```
N = M = 4, 6, 8, 10
P = 1
```

All matrices and scratch memory should not overlap and be aligned on 8-bytes boundary

2.8.6 Cholesky Backward Substitution for Pseudo-inversion

Description

These functions make backward substitution stage of pseudo-inversion. They use Cholesky decomposition of original matrices and results of forward substitution.

Precision

2 variants available:

Type	Description
f	floating point input, floating point output
32x32	32-bit fixed point

Algorithm

Solution of equations

$$R_{NxN} \cdot x_{NxP} = y_{NxP}$$

Prototype, floating point

```
void cbkw (void * pScr,
                  complex_float * x,
             const complex_float * R,
             const complex_float * D,
             const complex_float * y);

void  rbkw (void * pScr,
                   float32_t * x,
              const float32_t * R,
              const float32_t * D,
              const float32_t * y);
```

N	Function	API	Scratch allocation function
	Real data		
4	matcholbkwsubst4x4f	rbkw	matcholbkwsubst4x4f_getScratchSize
6	matcholbkwsubst6x6f	rbkw	matcholbkwsubst6x6f_getScratchSize
8	matcholbkwsubst8x8f	rbkw	matcholbkwsubst8x8f_getScratchSize
10	matcholbkwsubst10x10f	rbkw	matcholbkwsubst10x10f_getScratchSize
	Complex data		
4	cmatcholbkwsubst4x4f	cbkw	cmatcholbkwsubst4x4f_getScratchSize
6	cmatcholbkwsubst6x6f	cbkw	cmatcholbkwsubst6x6f_getScratchSize
8	cmatcholbkwsubst8x8f	cbkw	cmatcholbkwsubst8x8f_getScratchSize
10	cmatcholbkwsubst10x10f	cbkw	cmatcholbkwsubst10x10f_getScratchSize

Prototype, fixed point

```
void cbkw32 (void * pScr,
                  complex_fract32 * x,
             const complex_fract32 * R,
             const complex_fract32 * D,
             const complex_fract32 * y, int qXYR);

void  rbkw32 (void * pScr,
                   int32_t * x,
              const int32_t * R,
              const complex_fract32 * D,
              const int32_t * y, int qXYR);
```

N	Function	API	Scratch allocation function

	Real 32-bit data		
4	matcholbkwsubst4x4_32x32	rbkw32	matcholbkwsubst4x4_32x32_getScratchSize
6	matcholbkwsubst6x6_32x32	rbkw32	matcholbkwsubst6x6_32x32_getScratchSize
8	matcholbkwsubst8x8_32x32	rbkw32	matcholbkwsubst8x8_32x32_getScratchSize
10	matcholbkwsubst10x10_32x32	rbkw32	matcholbkwsubst10x10_32x32_getScratchSize
	Complex 32-bit data		
4	cmatcholbkwsubst4x4_32x32	cbkw32	cmatcholbkwsubst4x4_32x32_getScratchSize
6	cmatcholbkwsubst6x6_32x32	cbkw32	cmatcholbkwsubst6x6_32x32_getScratchSize
8	cmatcholbkwsubst8x8_32x32	cbkw32	cmatcholbkwsubst8x8_32x32_getScratchSize
10	cmatcholbkwsubst10x10_32x32	cbkw32	cmatcholbkwsubst10x10_32x32_getScratchSize

Arguments

Type	Name	Size	Description
Parameters			
int	N		Matrix dimension (number of columns in matrix A)
int	P		Number of columns in right-side matrix B
Input			
float32_t, complex_float, int32_t, complex_fract32	R	[((N+1)*N)/2]	Cholesky upper-triangle matrix R For fixed point API, the representation is Q(qR)
float32_t, complex_float, int32_t, complex_fract32	y	[N*P]	Results of forward substitution stage. For fixed point API, the representation is Q(qY)
float32_t, complex_float	D	[N]	reciprocals of main diagonal. NOTE: for the fixed point API, these data are stored internally in special format with separate mantissa and exponent for better accuracy and dynamic range control. So, even for the real data, they stored as pairs of 2 integers and packed to the complex_fract32 format
Output			
float32_t, complex_float, complex_fract32	x	[N*P]	Decision matrix x. For fixed point API, the representation is Q(qX)
int	qXYR		combination of fixed point representation (matrices R, x and y) qXYR=qX-qY+qR (for fixed point API only)
Temporary			
	pScr		Scratch memory

Returned value

none

Restrictions

```
N = 4, 6, 8, 10
P = 1
```

All matrices and scratch should not overlap and be aligned on 8-bytes boundary

2.8.7 Cholesky Presudoinversion

Description

Obtain Left Inverse of a matrix using Cholesky Decomposition

Precision

2 variants available:

Type	Description

f	floating point input, floating point output
32x32	32-bit fixed point

Algorithm

find $R_{NxN} = chol(A^*_{MxN} \cdot A_{MxN} + \sigma^2 I_{NxN})$: $R^*_{NxN} \cdot R_{NxN} \cong A^*_{MxN} \cdot A_{MxN} + \sigma^2 I_{NxN}$

solve $R^*_{NxN} \cdot x_{NxM} = A^*_{MxN}$

Prototypes, floating point

```
void  rinv (void* pScr,
       float32_t *x,
       const float32_t * A,
       const float32_t  sigma2);

void  cinv(void* pScr,
       complex_float *x,
       const complex_float * A,
       const float32_t  sigma2);
```

N, M	Function	API	Scratch allocation function
	Real data		
4	matcholpseudoinv4x4f	rinv	matcholpseudoinv4x4f_getScratchSize
6	matcholpseudoinv6x6f	rinv	matcholpseudoinv6x6f_getScratchSize
8	matcholpseudoinv8x8f	rinv	matcholpseudoinv8x8f_getScratchSize
10	matcholpseudoinv10x10f	rinv	matcholpseudoinv10x10f_getScratchSize
	Complex data		
4	cmatcholpseudoinv4x4f	cinv	cmatcholpseudoinv4x4f_getScratchSize
6	cmatcholpseudoinv6x6f	cinv	cmatcholpseudoinv6x6f_getScratchSize
8	cmatcholpseudoinv8x8f	cinv	cmatcholpseudoinv8x8f_getScratchSize
10	cmatcholpseudoinv10x10f	cinv	cmatcholpseudoinv10x10f_getScratchSize

Prototypes, fixed point

```
void  rinv (void* pScr,
       int32_t *x,
       const int32_t * A,
       const float32_t  sigma2);

void  cinv(void* pScr,
       complex_float *x,
       const complex_float * A,
       const float32_t  sigma2);
```

N, M	Function	API	Scratch allocation function
	Real data		
4	matcholpseudoinv4x4f	rinv	matcholpseudoinv4x4f_getScratchSize
6	matcholpseudoinv6x6f	rinv	matcholpseudoinv6x6f_getScratchSize
8	matcholpseudoinv8x8f	rinv	matcholpseudoinv8x8f_getScratchSize
10	matcholpseudoinv10x10f	rinv	matcholpseudoinv10x10f_getScratchSize
	Complex data		
4	cmatcholpseudoinv4x4f	cinv	cmatcholpseudoinv4x4f_getScratchSize
6	cmatcholpseudoinv6x6f	cinv	cmatcholpseudoinv6x6f_getScratchSize
8	cmatcholpseudoinv8x8f	cinv	cmatcholpseudoinv8x8f_getScratchSize
10	cmatcholpseudoinv10x10f	cinv	cmatcholpseudoinv10x10f_getScratchSize

Arguments

Type	Name	Size	Description
Parameters			
int	M		Matrix dimension (number of rows in matrix A)

int	N		Matrix dimension (number of columns in matrix A)
Input			
float32_t, complex_float	A	[M][N]	input matrix
float32_t	sigma2		regularization term
Output			
float32_t, complex_float	x	[N][M]	pseudoinverse matrix
Temporary			
	pScr		Scratch memory

Returned value

none

Restrictions

N = 4, 6, 8, 10

All matrices and scratch should not overlap and be aligned on 8-bytes boundary

2.8.8 Cholesky Preprocessing

Description

Preprocessing for Least Square Solutions

Precision

2 variants available:

Type	Description
f	floating point input, floating point output
32x32	32-bit fixed point

Algorithm

$$Z_{NxN} = A^*_{MxN} \cdot A_{MxN} + \sigma^2 I_{NxN}$$

Prototypes, floating point

```
void  rprep(void* pScr,
       float32_t *Z,
       const float32_t * A,
       float32_t  sigma2);

void  cprep(void* pScr,
       complex_float *Z,
       const complex_float * A,
       float32_t  sigma2);
```

N, M	Function	API	Scratch allocation function
	Real data, floating point		
4	matcholpreprocess4x4f	rprep	matcholpreprocess4x4f_getScratchSize
6	matcholpreprocess6x6f	rprep	matcholpreprocess6x6f_getScratchSize
8	matcholpreprocess8x8f	rprep	matcholpreprocess8x8f_getScratchSize
10	matcholpreprocess10x10f	rprep	matcholpreprocess10x10f_getScratchSize
	Complex data, floating point		
4	cmatcholpreprocess4x4f	cprep	cmatcholpreprocess4x4f_getScratchSize
6	cmatcholpreprocess6x6f	cprep	cmatcholpreprocess6x6f_getScratchSize
8	cmatcholpreprocess8x8f	cprep	cmatcholpreprocess8x8f_getScratchSize
10	cmatcholpreprocess10x10f	cprep	cmatcholpreprocess10x10f_getScratchSize

Prototypes, fixed point

```
void  rprep32(void* pScr,
       int64_t *Z,
       const int32_t * A,
       int32_t sigma2, int qRA);

void  cprep32(void* pScr,
       complex_fract64 *Z,
       const complex_fract32 * A,
       int32  sigma2, int qRA);
```

N, M	Function	API	Scratch allocation function
	Real 32-bit data		
4	matcholpreprocess4x4_32x32	rprep32	matcholpreprocess4x4_32x32_getScratchSize
6	matcholpreprocess6x6_32x32	rprep32	matcholpreprocess6x6_32x32_getScratchSize
8	matcholpreprocess8x8_32x32	rprep32	matcholpreprocess8x8_32x32_getScratchSize
10	matcholpreprocess10x10_32x32	rprep32	matcholpreprocess10x10_32x32_getScratchSize
	Complex 32-bit data		
4	cmatcholpreprocess4x4_32x32	cprep32	cmatcholpreprocess4x4_32x32_getScratchSize
6	cmatcholpreprocess6x6_32x32	cprep32	cmatcholpreprocess6x6_32x32_getScratchSize
8	cmatcholpreprocess8x8_32x32	cprep32	cmatcholpreprocess8x8_32x32_getScratchSize
10	cmatcholpreprocess10x10_32x32	cprep32	cmatcholpreprocess10x10_32x32_getScratchSize

Arguments

Type	Name	Size	Description
Parameters			
int	M		Matrix dimension (number of rows in matrix A)
int	N		Matrix dimension (number of columns in matrix A)
Input			
float32_t, complex_float, int32_t, complex_fract32	A	[M][N]	input matrix, For the fixed point, the representation is Q(qA)
float32_t, int32_t	sigma2		Regularization term. . For fixed point, the representation should be Q(2*qA-30)
int	qRA		qR-qA (for the fixed point API only); difference between fixed point representations of R and A matrices (for the fixed point API only). Should be equal or less than 0 (typically -2)
Output			
float32_t, complex_float, int64_t, complex_fract64	Z	[N][N]	output matrix. For the fixed poiont, the representation is Q(2*qR-4)
Temporary			
	pScr		Scratch memory

Returned value

none

Restrictions

N = 4, 6, 8, 10

All matrices and scratch should not overlap and be aligned on 8-bytes boundary

2.9 Fitting/Interpolation

2.9.1 Polynomial Approximation

Description

Functions calculate polynomial approximation for all values from given vector. Fixed point functions take polynomial coefficients in Q31 precision.
NOTE:
approximation is calculated like Taylor series that is why overflow may potentially occur if cumulative sum of coefficients given from the last to the first coefficient is bigger that 1. To avoid this negative effect for fixed point routines, all the coefficients may be scaled down and result will be shifted left after all intermediate computations. Amount of this left shift is controlled by `lsh` argument.

Precision

2 variants available:

Type	Description
32x32	32-bit inputs, 32-bit coefficients, 32-bit output.
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$z_n = \sum_{m=0}^{M} c_m x_n^m, n = \overline{0 \ldots N-1}$$

Prototype

```
void vec_poly8f
             (float32_t *  z, const float32_t *  x,
              const float32_t *  c, int N );
void vec_poly4_32x32
             (int32_t *  z, const int32_t *  x,
              const int32_t *  c, int lsh, int N );
void vec_poly8_32x32
              (int32_t *  z, const int32_t *  x,
               const int32_t *  c, int lsh, int N );
void vec_poly4f
             (float32_t *  z, const float32_t *  x,
              const float32_t *  c, int N );
```

Arguments

Type	Name	Size	Description
Input			
`int32_t, float32_t`	`x`	`N`	input data, Q31 or floating point
`int32_t, float32_t`	`c`	`5 or 9`	coefficients (5 coefficients for `vec_poly4_xxx` and 9 coefficients for `vec_poly8_xxx`), Q31 or floating point
`int`	`lsh`		additional left shift for result (for fixed point routines only)
`int`	`N`		length of vectors
Output			
`int32_t, float32_t`	`z`	`N`	result, Q31 or floating point

Returned value

None

Restrictions

`x,c,z` should not overlap

Conditions for optimum performance

`x,c,z` - aligned on 16-byte boundary
`N` - multiple of 2

2.10 Fast Fourier Transforms

FFT functions make floating point, 32x32, 32x16, 16x16-bit scaling fast Fourier transforms for complex/real data. Also, they use bit-reversal permutations so spectral data appear in the usual order. They normally use in-place transformations so **input data may be damaged**.

Different types if data scaling are provided by FFT functions. For all types of scaling, the internal representation of the data is the same as the input/output data. For these functions, the internal representation of the data is `complex_fract32`.

Basic scaling modes:

Scaling option	Scaling mode	Data processing	Performance & Accuracy
0	no scaling	In this mode, the FFT never scales the intermediate data.So, this mode is used only in very specific cases where this behavior is acceptable, i.e. in 2D DCT for JPEG, see 2.10.12	the best cycles, but worse accuracy on data with small magnitude and may suffer from overflow in the intermediate stages
2	dynamic scaling	The input data are normalized in the first phase of the FFT, but so that there is no overflow. In subsequent phases, the data are automatically shifted to the right, so that there is no overflow. The function returns a total shift count, which can be negative under certain conditions (i.e. weak input signals).	the best accuracy, but worse performance compared with static scaling
3	static scaling	The data are shifted to the right before each FFT phase, the amount of shift is independent of the input data and is chosen so that there is no overflow for any input data.	better performance but worse accuracy compared with dynamic scaling

FFT/IFFT functions family with improved memory efficiency (`fft_cplx<prec>_ie`, `fft_real<prec>_ie`) as well as floating point FFT functions[2] expose smaller program- and constant data memory footprint. They differ from regular FFT/IFFT functions in the following aspects:

- cycles performance is compromised in favor of memory efficiency
- twiddle factor tables are provided by user. A single table may be shared between FFTs/IFFTs of varying size (see para 4.2)

All fixed-point FFT functions (including scaling and non-scaling) return total number of right shifts (`t`) occurred during all stages. Floating point FFTs do not make additional scaling so they always return 0 to indicate this fact. So, FFT/IFFT output will be scaled by 2^t. Library functions from 2.5.5help to convert results to desired scale or Q-representation. In these computations you have to take into account the fact that FFT→IFFT chain amplifies signal by the length of FFT `N` for complex transforms and by `N/2` for real transforms.
For example, consider processing chain:

[2] Floating point FFT available only with improved memory efficiency API

`y=FFT(x) → w=some_processing(y) → z=IFFT(w)` where `N` is the length of FFT, FFT returns total shift amount t_{FFT} and IFFT returns t_{IFFT}.
To move `z` to the same scale as `x` you have to shift it by:
$t_{FFT} + t_{IFFT} + \log_2(N) \equiv t_{FFT} + t_{IFFT} + 31 - scl_bexp32(N)$

Alternatively, you may treat it as changing Q-representation. For example, DCT functions (with length 32) always return total number of shifts equals to $\log_2(32)=5$. So, if its input is Q31, output will be in Q26.

The table below summarizes how number of right shifts depends on selected scaled option.

Scaling option	FFT functions family	Returned number of right shifts
0	2D DCT	0
2	all FFT functions	depends on input data
3	FFT/IFFT on complex data	`log2(N)+1`
3	FFT/IFFT on real data, DCT	`log2(N)+1`

There are limited combinations of precision, scaling options and restrictions on the dynamic range of the input signal available:

Precision	Scaling options	Restrictions on the dynamic range of the input signal
FFT/IFFT		
`cplx32x16`	2 – 32-bit dynamic scaling 3 – fixed scaling before each stage	None
`cplx32x32`	2 – 32-bit dynamic scaling 3 – fixed scaling before each stage	None
`cplx16x16`	2 – 16-bit dynamic scaling 3 – fixed scaling before each stage	None
`cplx16x16_ie`	2 – 16-bit dynamic scaling	None
`cplx32x16_ie`	2 – 32-bit dynamic scaling 3 – fixed scaling before each stage	None
`cplx32x32_ie`	2 – 32-bit dynamic scaling 3 – fixed scaling before each stage	None
`real32x16`	2 – 32-bit dynamic scaling 3 – fixed scaling before each stage	None
`real32x32`	2 – 32-bit dynamic scaling 3 – fixed scaling before each stage	None
`real16x16`	2 – 16-bit dynamic scaling 3 – fixed scaling before each stage	None
`real16x16_ie`	2 – 16-bit dynamic scaling	None
`real32x16_ie`	2 – 32-bit dynamic scaling 3 – fixed scaling before each stage	None
`real32x32_ie`	2 – 32-bit dynamic scaling 3 – fixed scaling before each stage	None
DCT		
`dct_32x16`, `dct_32x32`, `dct_16x16`, `dct4_32x16`, `dct4_32x32`, `mdct_32x16`, `mdct_32x32`, `imdct_32x16`, `imdct_32x32`	3 – fixed scaling before each stage	None
`dct2d_8x16`	0 – no scaling	None

Precision	Scaling options	Restrictions on the dynamic range of the input signal
idct2d_16x8	0 – no scaling	None

2.10.1 FFT on Complex Data

Description

These functions make FFT on complex data.
NOTES:
1. Bit-reversing permutation is done here.
2. FFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after the call
3. 32x32, 32x16,16x16 FFT supports mixed radix transforms

Precision

3 variants available:

Type	Description
32x16	32-bit input/outputs, 16-bit twiddles
32x32	32-bit input/outputs, 32-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles

Algorithm

$$y = FFT(x)$$

Prototype

```
int fft_cplx32x16(
              int32_t* y, int32_t* x, fft_handle_t h, int scalingOption)
int fft_cplx32x32(
              int32_t* y, int32_t* x, fft_handle_t h, int scalingOption)
int fft_cplx16x16(
              int16_t* y, int16_t* x, fft_handle_t h, int scalingOption)
```

FFT handles:

N	32x16	32x16	32x32
16	cfft16_16	cfft16_16	cfft32_16
32	cfft16_32	cfft16_32	cfft32_32
64	cfft16_64	cfft16_64	cfft32_64
128	cfft16_128	cfft16_128	cfft32_128
256	cfft16_256	cfft16_256	cfft32_256
512	cfft16_512	cfft16_512	cfft32_512
1024	cfft16_1024	cfft16_1024	cfft32_1024
2048	cfft16_2048	cfft16_2048	cfft32_2048
4096	cfft16_4096	cfft16_4096	cfft32_4096

FFT handles for mixed radix transforms (for 16x16, 32x16 only):

N	16x16	N	32x16
160	cnfft16_160	160	cnfft32x16_160
192	cnfft16_192	192	cnfft32x16_192
240	cnfft16_240	240	cnfft32x16_240
320	cnfft16_320	320	cnfft32x16_320
384	cnfft16_384	384	cnfft32x16_384
480	cnfft16_480	480	cnfft32x16_480

FFT handles for mixed radix transforms (for 32x32 only):

N	32x32	N	32x32	N	32x32
12	cnfft32_12	144	cnfft32_144	360	cnfft32_360
24	cnfft32_24	160	cnfft32_160	384	cnfft32_384
36	cnfft32_36	180	cnfft32_180	400	cnfft32_400
48	cnfft32_48	192	cnfft32_192	432	cnfft32_432
60	cnfft32_60	200	cnfft32_200	480	cnfft32_480

72	cnfft32_72	216	cnfft32_216	540	cnfft32_540
80	cnfft32_80	240	cnfft32_240	576	cnfft32_576
96	cnfft32_96	288	cnfft32_288	600	cnfft32_600
100	cnfft32_100	300	cnfft32_300	768	cnfft32_768
108	cnfft32_108	320	cnfft32_320	960	cnfft32_960
120	cnfft32_120	324	cnfft32_324		

, where N - FFT size

Arguments

Type	Name	Size	Description
Input			
int32_t or int16_t	x	2*N	complex input signal. Real and imaginary data are interleaved and real data goes first
fft_handle_t	h		handle to specific FFT tables
int	scalingOption		scaling option (see table in para 2.10)
Output			
int32_t or int16_t	y	2*N	output spectrum. Real and imaginary data are interleaved and real data goes first

Returned value total number of right shifts occurred during scaling procedure

Restrictions x,y should not overlap
x,y aligned on a 16-bytes boundary

2.10.2 FFT on Real Data

Description These functions make FFT on real data forming half of spectrum
NOTES:
1. Bit-reversing reordering is done here.
2. FFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after the call.
3. Real data FFT function calls fft_cplx() to apply complex FFT of size N/2 to input data and then transforms the resulting spectrum.
4. 32x32, 32x16,16x16 FFT supports mixed radix transforms

Precision 3 variants available:

Type	Description
32x16	32-bit input/outputs, 16-bit twiddles
32x32	32-bit input/outputs, 32-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles

Algorithm $y = FFT(real(x))$

Prototype

```
int fft_real32x16(
              int32_t* y, int32_t* x, fft_handle_t h, int scalingOpt)
int fft_real32x32(
              int32_t* y, int32_t* x, fft_handle_t h, int scalingOpt)
int fft_real16x16(
              int16_t* y, int16_t* x, fft_handle_t h, int scalingOpt)
```

FFT handles:

N	32x16	32x16	32x32
32	rfft16_32	rfft16_32	rfft32_32
64	rfft16_64	rfft16_64	rfft32_64
128	rfft16_128	rfft16_128	rfft32_128
256	rfft16_256	rfft16_256	rfft32_256
512	rfft16_512	rfft16_512	rfft32_512
1024	rfft16_1024	rfft16_1024	rfft32_1024
2048	rfft16_2048	rfft16_2048	rfft32_2048
4096	rfft16_4096	rfft16_4096	rfft32_4096
8192	rfft16_8192	rfft16_8192	rfft32_8192

FFT handles for mixed radix transforms (for 16x16, 32x16 only) :

N	16x16	N	32x16
160	rnfft16_160	160	rnfft32x16_160
192	rnfft16_192	192	rnfft32x16_192
240	rnfft16_240	240	rnfft32x16_240
320	rnfft16_320	320	rnfft32x16_320
384	rnfft16_384	384	rnfft32x16_384
480	rnfft16_480	480	rnfft32x16_480

FFT handles for mixed radix transforms (for 32x32 only) :

N	32x32	N	32x32	N	32x32
12	rnfft32_12	160	rnfft32_160	480	rnfft32_480
24	rnfft32_24	180	rnfft32_180	540	rnfft32_540
30	rnfft32_30	192	rnfft32_192	576	rnfft32_576
36	rnfft32_36	216	rnfft32_216	720	rnfft32_720
48	rnfft32_48	240	rnfft32_240	768	rnfft32_768
60	rnfft32_60	288	rnfft32_288	960	rnfft32_960
72	rnfft32_72	300	rnfft32_300	1152	rnfft32_1152
90	rnfft32_90	320	rnfft32_320	1440	rnfft32_1440
96	rnfft32_96	324	rnfft32_324	1536	rnfft32_1536
108	rnfft32_108	360	rnfft32_360	1920	rnfft32_1920
120	rnfft32_120	384	rnfft32_384		
144	rnfft32_144	432	rnfft32_432		

, where N - FFT size

Arguments

Type	Name	Size	Description
Input			
int32_t or int16_t	x	N	input signal
fft_handle_t	h		handle to specific FFT tables
int	scalingOpt		scaling option (see table in para 2.10)
Output			
int32_t or int16_t	y	(N/2+1)*2	output spectrum (positive side). Real and imaginary data are interleaved and real data goes first

Returned value

total number of right shifts occurred during scaling procedure

Restrictions

Arrays should not overlap
x,y - aligned on a 16-bytes boundary

2.10.3 Inverse FFT on Complex Data

Description

These functions make inverse FFT on complex data.
NOTES:
1. Bit-reversing reordering is done here.
2. FFT runs in-place algorithm so **INPUT DATA WILL APPEAR DAMAGED** after call
3. 32x32, 32x16,16x16 FFT supports mixed radix transforms

Precision

3 variants available:

Type	Description
32x16	32-bit input/outputs, 16-bit twiddles
32x32	32-bit input/outputs, 32-bit twiddles

16x16	16-bit input/outputs, 16-bit twiddles

Algorithm

$$y = FFT^{-1}(x)$$

Prototype

```
int ifft_cplx32x16(
                int32_t * y, int32_t* x, fft_handle_t h, int scalingOption)
int ifft_cplx32x32(
                int32_t * y, int32_t* x, fft_handle_t h, int scalingOption)
int ifft_cplx16x16(
                int16_t* y, int16_t* x, fft_handle_t h, int scalingOption)
```

FFT handles:

N	32x16	32x16	32x32
16	cifft16_16	cifft16_16	cifft32_16
32	cifft16_32	cifft16_32	cifft32_32
64	cifft16_64	cifft16_64	cifft32_64
128	cifft16_128	cifft16_128	cifft32_128
256	cifft16_256	cifft16_256	cifft32_256
512	cifft16_512	cifft16_512	cifft32_512
1024	cifft16_1024	cifft16_1024	cifft32_1024
2048	cifft16_2048	cifft16_2048	cifft32_2048
4096	cifft16_4096	cifft16_4096	cifft32_4096

FFT handles for mixed radix transforms (for 16x16, 32x16 only):

N	16x16	N	32x16
160	cinfft16_160	160	cinfft32x16_160
192	cinfft16_192	192	cinfft32x16_192
240	cinfft16_240	240	cinfft32x16_240
320	cinfft16_320	320	cinfft32x16_320
384	cinfft16_384	384	cinfft32x16_384
480	cinfft16_480	480	cinfft32x16_480

FFT handles for mixed radix transforms (for 32x32 only):

N	32x32	N	32x32	N	32x32
12	cinfft32_12	144	cinfft32_144	360	cinfft32_360
24	cinfft32_24	160	cinfft32_160	384	cinfft32_384
36	cinfft32_36	180	cinfft32_180	400	cinfft32_400
48	cinfft32_48	192	cinfft32_192	432	cinfft32_432
60	cinfft32_60	200	cinfft32_200	480	cinfft32_480
72	cinfft32_72	216	cinfft32_216	540	cinfft32_540
80	cinfft32_80	240	cinfft32_240	576	cinfft32_576
96	cinfft32_96	288	cinfft32_288	600	cinfft32_600
100	cinfft32_100	300	cinfft32_300	768	cinfft32_768
108	cinfft32_108	320	cinfft32_320	960	cinfft32_960
120	cinfft32_120	324	cinfft32_324		

`, where N` – IFFT size

Arguments

Type	Name	Size	Description
Input			
int32_t or int16_t	x	2*N	input spectrum. Real and imaginary data are interleaved and real data goes first
fft_handle_t	h		handle to specific FFT tables
int	scalingOpt		scaling option (see table in para 2.10)
Output			
int32_t or	y	2*N	complex output signal. Real and imaginary data are

int16_t			interleaved and real data goes first

Returned value total number of right shifts occurred during scaling procedure

Restrictions `x,y` - should not overlap
`x,y` - aligned on 16-bytes boundary

2.10.4 Inverse FFT Forming Real Data

Description These functions make inverse FFT on half spectral data forming real data samples
NOTES:
1. Bit-reversing reordering is done here.
2. IFFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after call.
3. Inverse FFT function for real signal transforms the input spectrum and then calls `ifft_cplx()` with FFT size set to `N/2`. . 32x32, 32x16,16x16 FFT supports mixed radix transforms

Precision 3 variants available:

Type	Description
32x16	32-bit input/outputs, 16-bit twiddles
32x32	32-bit input/outputs, 32-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles

Algorithm $y = real(FFT^{-1}(x))$

Prototype

```
int ifft_real32x16(
                int32_t * y, int32_t* x, fft_handle_t h, int scalingOpt)
int ifft_real32x32(
                int32_t * y, int32_t* x, fft_handle_t h, int scalingOpt)
int ifft_real16x16(
                int16_t * y, int16_t* x, fft_handle_t h, int scalingOpt)
```

FFT handles:

N	32x16	32x16	32x32
32	rifft16_32	rifft16_32	rifft32_32
64	rifft16_64	rifft16_64	rifft32_64
128	rifft16_128	rifft16_128	rifft32_128
256	rifft16_256	rifft16_256	rifft32_256
512	rifft16_512	rifft16_512	rifft32_512
1024	rifft16_1024	rifft16_1024	rifft32_1024
2048	rifft16_2048	rifft16_2048	rifft32_2048
4096	rifft16_4096	rifft16_4096	rifft32_4096
8192	rifft16_8192	rifft16_8192	rifft32_8192

FFT handles for mixed radix transforms (for 16x16, 32x16 only):

N	16x16	N	32x16
160	rinfft16_160	160	rinfft32x16_160
192	rinfft16_192	192	rinfft32x16_192
240	rinfft16_240	240	rinfft32x16_240
320	rinfft16_320	320	rinfft32x16_320
384	rinfft16_384	384	rinfft32x16_384
480	rinfft16_480	480	rinfft32x16_480

FFT handles for mixed radix transforms (for 32x32 only):

N	32x32	N	32x32	N	32x32
12	rinfft32_12	160	rinfft32_160	480	rinfft32_480
24	rinfft32_24	180	rinfft32_180	540	rinfft32_540
30	rinfft32_30	192	rinfft32_192	576	rinfft32_576
36	rinfft32_36	216	rinfft32_216	720	rinfft32_720

48	rinfft32_48	240	rinfft32_240	768	rinfft32_768
60	rinfft32_60	288	rinfft32_288	960	rinfft32_960
72	rinfft32_72	300	rinfft32_300	1152	rinfft32_1152
90	rinfft32_90	320	rinfft32_320	1440	rinfft32_1440
96	rinfft32_96	324	rinfft32_324	1536	rinfft32_1536
108	rinfft32_108	360	rinfft32_360	1920	rinfft32_1920
120	rinfft32_120	384	rinfft32_384		
144	rinfft32_144	432	rinfft32_432		

Arguments

`,where N` - IFFT size

Type	Name	Size	Description
Input			
int32_t or int16_t	x	(N/2+1)*2	input spectrum. Real and imaginary data are interleaved and real data goes first. The imaginary part of 0th and (N/2)th elements should be equal to zero.
fft_handle_t	h		handle to specific FFT tables
int	scalingOpt		scaling option (see table in para 2.10)
Output			
int32_t or int16_t	y	N	real output signal

Returned value

total number of right shifts occurred during scaling procedure

Restrictions

`x,y` should not overlap
`x,y` - aligned on 16-bytes boundary
x[(0)*2+1],
x[(N/2)*2+1] - should be equal to zero

2.10.5 FFT on Complex Data with Optimized Memory Usage

Description

These functions make FFT on complex data with optimized memory usage
NOTES:
1. Bit-reversing permutation is done here.
2. FFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after the call
3. FFT of size `N` may be supplied with constant data (twiddle factors) of a larger-sized FFT `= N*twdstep`.
4. stereo FFTs accept inputs from and provides outputs in interleaved order: left complex sample, right complex sample

Precision

4 variants available:

Type	Description
32x16	32-bit input/outputs, 16-bit twiddles. Ordinary (single channel) variant and stereo
32x32	32-bit input/outputs, 32-bit twiddles. Ordinary (single channel) variant and stereo
16x16	16-bit input/outputs, 16-bit twiddles. Ordinary (single channel) variant and stereo
f	floating point. Requires VFPU/SFPU core option. Ordinary variant (single channel) and stereo

Algorithm

$$y = FFT(x)$$

Prototype (single channel variants)

```
int fft_cplx32x16_ie(
              complex_fract32* y, complex_fract32* x,
              const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int fft_cplx32x32_ie(
              complex_fract32* y, complex_fract32* x,
              const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int fft_cplx16x16_ie(
              complex_fract16* y, complex_fract16* x,
              const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int fft_cplxf_ie (
```

```
                complex_float * y, complex_float * x,
                const complex_float* twd,
                int twdstep, int N );
```

Prototype (stereo variants)

```
int stereo_fft_cplx32x16_ie(
                complex_fract32* y, complex_fract32* x,
                const complex_fract16* twd,
                int twdstep, int N, int scalingOpt);
int stereo_fft_cplx32x32_ie(
                complex_fract32* y, complex_fract32* x,
                const complex_fract32* twd,
                int twdstep, int N, int scalingOpt);
int stereo_fft_cplx16x16_ie(
                complex_fract16* y, complex_fract16* x,
                const complex_fract16* twd,
                int twdstep, int N, int scalingOpt);
int stereo_fft_cplxf_ie (
                complex_float * y, complex_float * x,
                const complex_float* twd,
                int twdstep, int N );
```

Arguments

Type	Name	Size	Description
Input			
complex_fract16, complex_fract32, complex_float	x	N*S	complex input signal. Real and imaginary data are interleaved and real data goes first
complex_fract32, complex_fract16, complex_float	twd	N*3/4*twdstep	twiddle factor table of a complex-valued FFT of size N*twdstep
int	twdstep		twiddle step
int	N		FFT size
int	scalingOpt		scaling option (see table in para 2.10) , not applicable to the floating point function
Parameter			
	S		stereo option, 1 for ordinary (single channel) FFT, 2 - for stereo input/outputs. Note: it does not come in parameter list of functions, but just specifies the total size of input/output data
Output			
complex_fract16, complex_fract32, complex_float	y	N*S	output spectrum. Real and imaginary data are interleaved and real data goes first

Returned value

total number of right shifts occurred during scaling procedure. Floating function always return 0

Restrictions

x,y should not overlap
x,y - aligned on a 16-bytes boundary

2.10.6 FFT on Real Data with Optimized Memory Usage

Description

These functions make FFT on real data forming half of spectrum with optimized memory usage
NOTES:
1. Bit-reversing reordering is done here.
2. FFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after the call.
3. FFT of size N may be supplied with constant data (twiddle factors) of a larger-sized FFT = N*twdstep

Precision

4 variants available:

Type	Description
32x16	32-bit input/outputs, 16-bit twiddles

32x32	32-bit input/outputs, 32-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$y = FFT(real(x))$$

Prototype

```
int fft_real32x16_ie(
              complex_fract32* y, int32_t* x, const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int fft_real32x32_ie(
              complex_fract32* y, int32_t* x, const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int fft_real16x16_ie(
              complex_fract16* y, int16_t* x, const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int fft_realf_ie(
              complex_float* y, float32_t* x, const complex_float* twd,
              int twdstep, int N);
```

Arguments

Type	Name	Size	Allocated Size	Description
Input				
int16_t, int32_t, float32_t	x	N	N	input signal
complex_fract32, complex_fract16, complex_float	twd	N*3/4 *twdstep		twiddle factor table of a complex-valued FFT of size N*twdstep
int	twdstep			twiddle step
int	N			FFT size
int	scalingOpt			scaling option (see table in para 2.10) , not applicable to the floating point function
Output				
complex_fract16, complex_fract32, complex_float	y	N/2+1	N/2+1	output spectrum (positive side). Real and imaginary data are interleaved and real data goes first

Returned value

total number of right shifts occurred during scaling procedure. Floating function always return 0

Restrictions

Arrays should not overlap
x,y - aligned on a 16-bytes boundary

2.10.7 Inverse FFT on Complex Data with Optimized Memory Usage

Description

These functions make inverse FFT on complex data with optimized memory usage
NOTES:
1. Bit-reversing permutation is done here.
2. FFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after the call
3. FFT of size N may be supplied with constant data (twiddle factors) of a larger-sized FFT = N*twdstep.
4. stereo FFTs accept inputs from and provides outputs in interleaved order: left complex sample, right complex sample

Precision

4 variants available:

Type	Description
32x16	32-bit input/outputs, 16-bit twiddles. Ordinary (single channel) variant and stereo
32x32	32-bit input/outputs, 32-bit twiddles. Ordinary (single channel) variant and stereo
16x16	16-bit input/outputs, 16-bit twiddles. Ordinary (single channel) variant and stereo
f	floating point. Requires VFPU/SFPU core option. Ordinary (single channel) variant and stereo

Algorithm

$$y = FFT^{-1}(x)$$

Prototype (single channel variant)

```
int ifft_cplx32x16_ie(
              complex_fract32* y, complex_fract32* x,
              const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int ifft_cplx32x32_ie(
              complex_fract32* y, complex_fract32* x,
              const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int ifft_cplx16x16_ie(
              complex_fract16* y, complex_fract16* x,
              const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int ifft_cplxf_ie(
              complex_float* y, complex_float * x,
              const complex_float * twd,
              int twdstep, int N);
```

Prototype (streo variants)

```
int stereo_ifft_cplx32x16_ie(
              complex_fract32* y, complex_fract32* x,
              const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int stereo_ifft_cplx32x32_ie(
              complex_fract32* y, complex_fract32* x,
              const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int stereo_ifft_cplx16x16_ie(
              complex_fract16* y, complex_fract16* x,
              const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int stereo_ifft_cplxf_ie(
              complex_float* y, complex_float * x,
              const complex_float * twd,
              int twdstep, int N);
```

Arguments

Type	Name	Size	Description
Input			
complex_fract16, complex_float complex_fract32, complex_float	x	N*S	complex input signal. Real and imaginary data are interleaved and real data goes first
complex_fract32, complex_fract16, complex_float	twd	N*3/4*twdstep	twiddle factor table of a complex-valued FFT of size N*twdstep
int	twdstep		twiddle step
int	N		FFT size
int	scalingOpt		scaling option (see table in para 2.10) , not applicable to the floating point function
Parameter			
	S		stereo option, 1 for ordinary (single channel) FFT, 2 - for stereo input/outputs. Note: it does not come in parameter list of functions, but just specifies the total size of input/output data
Output			
complex_fract16, complex_fract32, complex_float	y	N*S	output spectrum. Real and imaginary data are interleaved and real data goes first

Returned value

total number of right shifts occurred during scaling procedure. Floating function always return 0

Restrictions

x,y should not overlap
x,y - aligned on a 16-bytes boundary

2.10.8 Inverse FFT on Real Data with Optimized Memory Usage

Description

These functions make inverse FFT on real data from half of spectrum with optimized memory usage
NOTES:
1. Bit-reversing reordering is done here.
2. FFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after the call.
3. FFT functions use input and output buffers for temporary storage of intermediate 32-bit data, so FFT functions with 24-bit packed I/O (`Nx3`-byte data) require that the buffers are large enough to keep `Nx4`-byte data.
4. FFT of size `N` may be supplied with constant data (twiddle factors) of a larger-sized FFT = `N*twdstep`

Precision

4 variants available:

Type	Description
`32x16`	32-bit input/outputs, 16-bit twiddles
`32x32`	32-bit input/outputs, 32-bit twiddles
`16x16`	16-bit input/outputs, 16-bit twiddles
`f`	floating point. Requires VFPU/SFPU core option

Algorithm

$$y = real(FFT^{-1}(x))$$

Prototype

```
int ifft_real32x16_ie(
              int32_t* y, complex_fract32* x, const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int ifft_real32x32_ie(
              int32_t* y, complex_fract32* x, const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int ifft_real16x16_ie(
              int32_t* y, complex_fract16* x, const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int ifft_realf_ie(
              float32_t* y, complex_float * x, const complex_float* twd,
              int twdstep, int N);
```

Arguments

Type	Name	Size	Allocated Size	Description
Input				
`complex_fract16`, `complex_fract32`, `complex_float`	`x`	`N/2+1`	`N/2+1`	input spectrum (positive side). Real and imaginary data are interleaved and real data goes first. The imaginary part of 0th and (N/2)th elements should be equal to zero.
`complex_fract32`, `complex_fract16`, `complex_float`	`twd`	`N*3/4*twdstep`		twiddle factor table of a complex-valued FFT of size `N*twdstep`
`int`	`twdstep`			twiddle step
`int`	`N`			FFT size
`int`	`scalingOpt`			scaling option (see table in para 2.10) , not applicable to the floating point function
Output				
`int16_t`, `int32_t`, `float32_t`	`y`	`N`	`N`	output real signal

Returned value

total number of right shifts occurred during scaling procedure. Floating function always return 0

Restrictions

Arrays should not overlap
`x,y` - aligned on a 16-bytes boundary
x[(0)*2+1],
x[(N/2)*2+1] - should be equal to zero

2.10.9 Power Spectrum

Description

These functions compute a normalized power spectrum from the output signal generated by an FFT function.
The `N` argument specifies the size of the FFT and must be a power of 2.
The `mode` argument is used to specify the type of FFT function used to generate the `x` array. If the `x` array has been generated from a frequency-domain complex input signal (output of complex FFT function), the `mode` argument must be set to 0. Otherwise the `mode` argument must be set to 1 to signify that the `x` array has been generated from a frequency-domain real input signal (output of real FFT function).
The `block_exponent` argument is used to control the normalization of the power spectrum. It will usually be set to the `block_exponent` that is returned by corresponding FFT functions. If the input array was generated by some other means, then the value specified for the `block_exponent` argument will depend upon how the FFT was calculated. If the function used to calculate the FFT did not scale the intermediate results at any of the stages of the computation, then set `block_exponent` to zero; if the FFT function scaled the intermediate results at each stage of the computation, then set `block_exponent` to `-1`; otherwise set `block_exponent` to the sum of negated base-2 logarithm of all scaling factors applied to data at intermediate FFT stages. This value will be in the range `0` to `log2(N)`.
`fft_spectrum` functions write the power spectrum to the output array `y`. If `mode` is set to `0`, then the length of the power spectrum will be `N`. If `mode` is set to `1`, then the length of the power spectrum will be `(N/2+1)`

Precision

3 variants available:

Type	Description
`16x32`	16-bit inputs, 32-bit outputs
`32x32`	32-bit inputs/outputs
`f`	floating point inputs/outputs. Requires VFPU/SFPU core option

Algorithm

For `mode` = 0 (`cfft`-generated input data):

$$y_n = \frac{\sqrt{x_n \cdot x_n^*}}{N}, n = \overline{0 \ldots N-1}$$

For `mode` = 1 (`rfft`-generated input data):

$$y_n = \frac{2 \cdot \sqrt{x_n \cdot x_n^*}}{N}, n = \overline{0 \ldots N/2}$$

Prototype

```
void fft_spectrumf     ( float32_t* y, const complex_float * x,
                         int N, int mode );
void fft_spectrum16x32 ( int32_t* y, const complex_fract16 * x,
                         int N, int block_exponent, int mode );
void fft_spectrum32x32 ( int32_t* y, const complex_fract32 * x,
                         int N, int block_exponent, int mode );
```

Arguments

Type	Name	Size	Description
Input			
`complex_fract16`, `complex_fract32`, `complex_float`	`x`	`N` (for `mode==0`) `N/2+1` (for `mode==1`)	input spectrum
`int`	`N`		FFT length
`int`	`block_exponent`		power spectrum normalization control
`int`	`mode`		power spectrum mode: 0 – complex signal 1 – real signal
Output			
`complex_fract16`, `complex_fract32`, `complex_float`	`y`	`N` (for `mode==0`) `N/2+1`	output power spectrum

		(for mode==1)	

Returned value none

Restrictions Arrays should not overlap
`x,y` - aligned on a 16-bytes boundary

2.10.10 Discrete Cosine Transform

Description These functions apply DCT (Type II, Type IV) to input
NOTE:
DCT runs in-place algorithm so **INPUT DATA WILL APPEAR DAMAGED** after the call.

Precision 4 variants available:

Type	Description
32x16	32-bit input/outputs, 16-bit twiddles
32x32	32-bit input/outputs, 32-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles
f	floating point. Requires VFPU/SFPU core option

Algorithm

DCT Type II: $y_{k=0..N-1} = \sum_{n=0}^{N-1} x_n \cdot \cos\left(\frac{\pi}{N} \cdot (n+0.5) \cdot k\right), n = \overline{0...N-1}$

DCT Type IV: $y_{k=0..N-1} = \sum_{n=0}^{N-1} x_n \cdot \cos\left(\frac{\pi}{N} \cdot (n+0.5) \cdot [k+0.5]\right), n = \overline{0...N-1}$

Prototype (DCT Type II)

```
int dct_32x16(int32_t* y, int32_t* x, dct_handle_t h, int scalingOpt);
int dct_32x32(int32_t* y, int32_t* x, dct_handle_t h, int scalingOpt);
int dct_16x16(int16_t* y, int16_t* x, dct_handle_t h, int scalingOpt);
int dctf     ( float32_t * y,float32_t * x, dct_handle_t h  );
```

DCT-II handles:

N	32x32	N	32x16, 16x16	N	floating point
32	`dct2_32_32`	32	`dct2_16_32`	32	`dct2_f_32`
64	`dct2_32_64`	64	`dct2_16_64`	64	`dct4_f_64`

`,where N` - DCT size

Arguments

Type	Name	Size	Description
Input			
`int16_t, int32_t, float32_t`	`x`	`N`	input signal
`dct_handle_t`	`h`		DCT-II handle
`int`	`scalingOpt`		scaling option (see table in para 2.10), not applicable to the floating point function
Output			
`int16_t, int32_t, float32_t`	`y`	`N`	output of transform

Prototype (DCT Type IV)

```
int dct4_32x16(int32_t* y, int32_t* x, dct_handle h, int scalingOpt);
int dct4_32x32(int32_t* y, int32_t* x, dct_handle h, int scalingOpt);
```

DCT-IV handles:

N	32x32	N	32x16

32	dct4_32_32	32	dct4_16_32
64	dct4_32_64	64	dct4_16_64
128	dct4_32_128	128	dct4_16_128
256	dct4_32_256	256	dct4_16_256
512	dct4_32_512	512	dct4_16_512

, where N - DCT size

Arguments

Type	Name	Size	Description
Input			
int32_t	x	N	input signal
dct_handle_t	h		DCT-IV handle
int	scalingOpt		scaling option (see table in para 2.10), not applicable to the floating point function
Output			
int16_t, int32_t, float32_t	y	N	output of transform

Returned value

total number of right shifts occurred during scaling procedure (0 for floating point function)

Restrictions

x,y should not overlap
x,y - aligned on 16-bytes boundary

2.10.11 Modified Discrete Cosine Transform

Description

These functions apply Modified DCT to input (convert 2N real data to N spectral components) and make inverse conversion forming 2N numbers from N inputs.

NOTE:
MDCT runs in-place algorithm so **INPUT DATA WILL APPEAR DAMAGED** after the call.

Precision

2 variants available:

Type	Description
32x16	32-bit input/outputs, 16-bit twiddles
32x32	32-bit input/outputs, 32-bit twiddles

Algorithm

MDCT: $y_{k=0..N-1} = \sum_{n=0}^{N-1} x_n \cdot \cos\left(\frac{\pi}{N} \cdot (n+0.5) \cdot k\right), n = \overline{0...N-1}$

Prototype (direct trasform)

```
int mdct_32x16(int32_t* y, int32_t* x, dct_handle_t h, int scalingOpt);
int mdct_32x32(int32_t* y, int32_t* x, dct_handle_t h, int scalingOpt);
```

MDCT/IMDCT handles:

N	32x32	N	32x16
32	mdct_32_32	32	mdct_16_32
64	mdct_32_64	64	mdct_16_64
128	mdct_32_128	128	mdct_16_128
256	mdct_32_256	256	mdct_16_256
512	mdct_32_512	512	mdct_16_512

, where N - MDCT/IMDCT size

Arguments

Type	Name	Size	Description
Input			
int32_t	x	N	input signal
dct_handle_t	h		MDCT handle

int	scalingOpt		scaling option (see table in para 2.10)
Output			
int32_t	y	2*N	output of transform

Algorithm

Inverse MDCT: $y_{k=0..N-1} = \sum_{n=0}^{N-1} x_n \cdot \cos\left(\frac{\pi}{N} \cdot (n+0.5) \cdot (k+0.5)\right), n = \overline{0..N-1}$

Prototype (inverse trasform)

```
int imdct_32x16(int32_t* y, int32_t* x, dct_handle h, int scalingOpt);
int imdct_32x32(int32_t* y, int32_t* x, dct_handle h, int scalingOpt);
```

Arguments

Type	Name	Size	Description
Input			
int32_t	x	2*N	input signal
dct_handle_t	h		IMDCT handle
int	scalingOpt		scaling option (see table in para 2.10)
Output			
int32_t	y	N	output of transform

Returned value

total number of right shifts occurred during scaling procedure (0 for floating point function)

Restrictions

`x,y` should not overlap
`x,y` - aligned on 16-bytes boundary

2.10.12 2D Discrete Cosine Transform

Description

These functions apply DCT (Type II) to the series of `L` input blocks of `NxN` pixels.

Precision

1 variant available:

Type	Description
8x16	8-bit unsigned input, 16-bit signed output

Algorithm

Algorithm uses ITU-T T.81 (JPEG compression) DCT-II definition with bias 128 and left-to-right, top-to-bottom orientation.

$$y_{n,m}^{(l)} = \frac{1}{4} C_m C_n \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (x_{j,i}^{(l)} - 128) \cos\frac{(2i+1)m\pi}{2N} \cos\frac{(2j+1)n\pi}{2N}, l = \overline{0..L-1}$$

$$C_k = \begin{cases} 1 & , k \neq 0 \\ 1/\sqrt{2} & , k \equiv 0 \end{cases}$$

$x_{j,i}^{(l)} = x[l \cdot N \cdot N + (N \cdot j + i)]$ - pixel from j-th row, i-th column from l-th NxN block

Prototype

```
int dct2d_8x16(int16_t* y, uint8_t * x, dct_handle_t h, int L, int scalingOpt);
```

2D-DCT handles:

N	8x16
8	dct2d_16_8

, where `N` - DCT size

Arguments

Type	Name	Size	Description
Input			
uint8_t	x	N*N*L	input pixels: `L` `NxN` blocks
dct_handle_t	h		DCT handle
int	L		number of input blocks
int	scalingOpt		scaling option (see table in para 2.10), should be 0
Output			

int16_t	y	N*N*L	output of transform: L NxN blocks

Returned value 0

Restrictions x,y should not overlap
x,y - aligned on 16-bytes boundary

2.10.13 2D Inverse Discrete Cosine Transform

Description These functions apply inverse DCT (Type II) to the series of L input blocks of NxN pixels.

Precision 1 variant available:

Type	Description
16x8	16-bit signed input, 8-bit unsigned output

Algorithm Algorithm uses ITU-T T.81 (JPEG compression) IDCT-II definition with bias 128 and left-to-right, top-to-bottom orientation.

$$y_{j,i}^{(l)} = 128 + \frac{1}{4}\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} C_m C_n x_{n,m}^{(l)} \cos\frac{(2i+1)m\pi}{2N}\cos\frac{(2j+1)n\pi}{2N}, l = \overline{0..L-1}$$

$$C_k = \begin{cases} 1 & ,k \neq 0 \\ 1/\sqrt{2} & ,k \equiv 0 \end{cases}$$

$x_{n,m}^{(l)} = x[l \cdot N \cdot N + (N \cdot n + m)]$ - sample from n-th row, m-th column from l-th 8x8 block

Prototype

```
int idct2d_16x8(uint8_t * y, int16_t* x, dct_handle_t h, int L, int scalingOpt);
```

2D-IDCT handles :

N	16x8
8	idct2d_16_8

,where N - IDCT size

Arguments

Type	Name	Size	Description
Input			
int16_t	x	N*N*L	input data: L NxN blocks
dct_handle_t	h		IDCT handle
int	L		number of input blocks
int	scalingOpt		scaling option (see table in para 2.10), should be 0
Output			
uint8_t	y	N*N*L	pixels: L NxN blocks

Returned value 0

Restrictions x,y should not overlap
x,y - aligned on 16-bytes boundary

2.11 Mel-Frequency Cepstral Coefficients

Mel-Frequency Cepstral Coefficients (MFCC) is an industry-standard representation of audio data for advanced audio processing, such as speech recognition software.

The MFCC package of the Library comprises the MFCC features extractor and its subsidiary component, the Log-scale Mel-frequency (LogMel) filterbank. The interconnection of these components and their operation are depicted in the figure below:

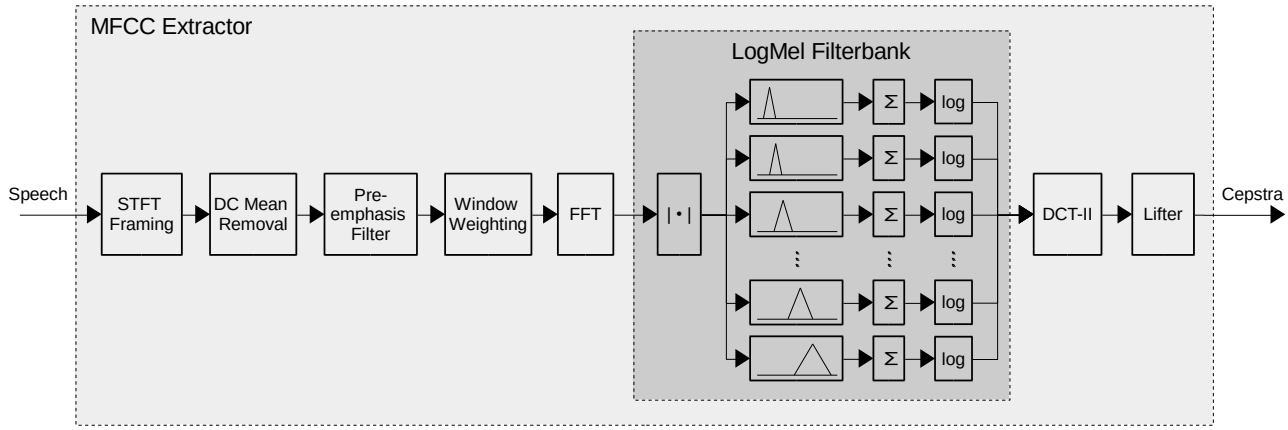


Figure 1. Block Diagram of the MFCC Extractor

Cepsral coefficient computation includes the following stages:

1. Input speech signal is optionally passed through a pre-emphasis filter (first order FIR).
2. Filtered signal is subject to short-time Fourier transform (STFT) followed by magnitude spectrum computation.
3. A set of filters is applied to the magnitude spectrum, with triangular weight functions constructed in such a way that the prescribed frequency range is divided into overlapping bands of equal mel-frequency width.
4. Log-scaled filterbank energies are decorrelated via a Discrete Cosine Transform Type II (DCT-II) to form cepstrum coefficients.
5. In the last step a sine lifter is optionally applied to cepstra to align coefficient magnitudes.

In general, the computation procedure follows the MFCC features extraction algorithm adopted in the Hidden Markov Models Toolkit (HTK), as descibed in:

S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, X. Liu, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, P. Woodland, The HTK Book (for HTK version 3.4), Cambridge University Engineering Department, 2009. http://htk.eng.cam.ac.uk/docs/docs.shtml

In addition, a number of options provide an ability to emulate the operation of another popular package for speech analysis:

The Auditory Toolbox for MATLAB by Malcolm Slaney, Version 2, Interval Research Corporation https://engineering.purdue.edu/~malcolm/interval/1998-010/

Besides of the primary use of the LogMel filterbank as an integral part of the MFCC extractor, it may be utilized on its own by means of a dedicated API, as shown below.

2.11.1 Compute Log Mel Filterbank Energies

Description

In the initialization stage, split the specified frequency range into 1/2 overlapping bands of equal mel-frequency width, and compute triangular weighting function for each band.
Data processing functions are applied to Fourier image of real signal, specified through the input argument `spectra[fftSize/2+1]`. Fourier image is converted to magnitude spectrum. For every mel-frequency band, magnitude samples are multiplied by the corresponding triangular weighting function, and summed together to form filterbank energies (FBEs). Finally, log-scaled FBEs are stored to the output argument `logFbe[mfbBandNum]`.

Precision

2 variants available:

Type	Description

32x32	32-bit fixed-point input/output data
f	Single precision floating-point input/output data

Algorithm

$$logFbe = \log\left(coeffMatrix \cdot |spectra|\right)$$

Parameters of log mel filterbank operation

```
typedef struct logmel_params_tag
{
```

Type	Name	Description
int	Fs	Sampling rate, Hz
int	fftSize	FFT size
fract32	mfbLowFreqQ8	Lowest band's left frequency edge, Hz (Q8).
fract32	mfbUppFreqQ8	Uppermost band's right frequency edge, Hz (Q8).
int	mfbBandNum	Number of mel filterbank spectral bands.
int	opt	Options to control various aspects of MFCC features extraction, ORed combination of `LOGMEL_OPT_<...>` flags (see the table below).

```
} logmel_params_t;
```

Option flags:

Name	Value	Description
LOGMEL_OPT_MELSCALE_HTK	0<<0	Use HTK mapping between linear and mel-scale frequencies.
LOGMEL_OPT_MELSCALE_AUDITORY	1<<0	Use Auditory Toolbox mapping between linear and mel-scale frequencies.
LOGMEL_OPT_FBELOG_NATURAL	0<<1	Compute base-e logarithm of filterbank energies (HTK).
LOGMEL_OPT_FBELOG_BASE10	1<<1	Compute base-10 logarithm of filterbank energies (Auditory).
LOGMEL_OPT_FBNORM_NONE	0<<2	No normalization of filterbank weight functions, peak at 1.0 (HTK).
LOGMEL_OPT_FBNORM_AREA	1<<2	Perform area normalization of filterbank weight functions (Auditory).

Object allocation

```
size_t logmel32x32_alloc( const logmel_params_t * params );
size_t logmelf_alloc    ( const logmel_params_t * params );
```

Type	Name	Description
Input		
logmel_params_t	params	Parameters of log mel filterbank operation

Returns: size of memory to be allocated for an instance object, in bytes, or zero if parameters do not satisfy the restrictions.

Object initialization

```
logmel32x32_handle_t logmel32x32_init( void * objmem,
                                       const logmel_params_t * params);
logmelf_handle_t     logmelf_init    ( void * objmem,
                                       const logmel_params_t * params );
```

Type	Name	Description
Input		
void *	objmem	Memory block allocated for the instance object
logmel_params_t	params	Parameters of log mel filterbank operation

Returns: handle to the object, or `NULL` if initialization failed.

Compute log mel filterbank energies

```
void logmel32x32_process( logmel32x32_handle_t handle, void * restrict pScr,
                          fract32 * restrict logFbe,
                          const complex_fract32 * restrict spectra,
                          int scaleExp );
void logmelf_process    ( logmelf_handle_t handle, void * restrict pScr,
                          float32_t * restrict logFbe,
                          const complex_float * restrict spectra,
```

```
int scaleExp );
```

Type	Name	Size	Description
Input			
complex_fract32, complex_float	spectra	fftSize/2+1	Fourier image of real signal, positive frequencies only; Q31 for 32x32
int	scaleExp		Exponent value to scale the Fourier image by a factor of $2^{scaleExp}$: **32x32** For full-scale Q31 real signal the scale exponent should be set to 15 plus the sum of bit shifts applied to data throughout the real-to-complex FFT transform, as indicated by the respective FFT routine. **f** For real signal varying in the range [-1,1] the scale exponent should be set to 15.
Output			
fract32, float32_t	logFbe	mfbBandNum	Log-scaled filterbank energies; Q6.25 for 32x32.
Temporary			
void*	pScr		Scratch memory area for the processing function. To determine the scratch area size, use the respective helper function: logmel<32x32|f> getScratchSize().

Determine size of the scratch memory area

```
size_t logmel32x32_getScratchSize( const logmel_params_t * params );
size_t logmelf_getScratchSize    ( const logmel_params_t * params );
```

Type	Name	Description
Input		
logmel_params_t	params	Parameters of log mel filterbank operation

Returns: size of scratch memory area, in bytes.

Restrictions

logFbe[],spectra[] – must not overlap, and must be aligned by 16-bytes
Fs – 8000 <= Fs <= 48000
fftSize – 256, 512, 1024 or 2048
mfbLowFreqQ8, mfbUppFreqQ8 – 0 <= mfbLowFreqQ8 < mfbUppFreqQ8 <= 16000*256
mfbBandNum – 0 < mfbBandNum <= 40

2.11.2 Compute Mel-Frequency Cepstrum Coefficients

Description

In the initialization stage, perform preliminary computations that allow efficient processing of audio stream in real-time environment:

1. Setup an internal instance of log mel filterbank.
2. Optionally invoke a user-supplied callback function to calculate STFT weighting window.
3. Compute a DCT Type II transform matrix.
4. Optionally pre-compute the sine lifter coefficients.

Audio stream is split into chunks of stftHopLen speech samples, specified through the input argument speech[stftHopLen]. Processing of a speech chunk encompasses the following steps:

1. Update the STFT sliding frame.
2. Estimate and subtract the DC mean from the source signal (optional).
3. Perform pre-emphasis filtering (optional).
4. Apply the STFT weighting window (optional).
5. Invoke a user-supplied callback function to perform the real-to-complex FFT.
6. Pass the resulting Fourier image to log mel filterbank to obtain log-scaled filterbank energies (LogFBEs).
7. Decorrelate LogFBEs through the DCT Type II transform, this forms the cepstral coefficients.
8. Align magnitudes of cepstral coefficients by the sine lifter (optional).

Resulting cepstral coefficients are strored to the output argument cepstra[cepstraNum].

Precision

2 variants available:

Type	Description
32x32	32-bit fixed-point input/output data
f	Single precision floating-point input/output data

Parameters of MFCC features extraction

```
typedef struct mfcc_params_tag
{
```

Type	Name	Default Value	Description
int	Fs	16000	Sampling rate, Hz
int	scaleExp	15	Specifies the scaling factor applied to speech signal: $2^{scaleExp}$.
fract16	preemph	31785	Pre-emphasis filter coefficient, Q15; set to 0 to disable the filter.
int	fftSize	512	FFT size
int	stftWinLen	400	Short-time Fourier transform window length, must not exceed `fftSize`.
int	stftHopLen	160	Number of audio samples between successive windows, must not exceed `stftWinLen`.
fract32	mfbLowFreqQ8	0	Lowest band's left frequency edge, Hz (Q8).
fract32	mfbUppFreqQ8	4000*256	Uppermost band's right frequency edge, Hz (Q8).
int	mfbBandNum	20	Number of mel filterbank spectral bands.
int	cepstraNum	12	Number of cepstral coefficients to compute, including the 0th coefficient.
int	lifter	22	Cepstral lifter parameter; zet to zero to disable the lifter.
int	opt	0	Options to control various aspects of MFCC features extraction, ORed combination of `MFCC_OPT_<...>` and `LOGMEL_OPT_<...>` flags (see the table below).

```
} mfcc_params_t;
```

Option flags:

Name	Value	Description
LOGMEL_OPT_MELSCALE_HTK	0<<0	Use HTK mapping between linear and mel-scale frequencies.
LOGMEL_OPT_MELSCALE_AUDITORY	1<<0	Use Auditory Toolbox mapping between linear and mel-scale frequencies.
LOGMEL_OPT_FBELOG_NATURAL	0<<1	Compute base-e logarithm of filterbank energies (HTK).
LOGMEL_OPT_FBELOG_BASE10	1<<1	Compute base-10 logarithm of filterbank energies (Auditory).
LOGMEL_OPT_FBNORM_NONE	0<<2	No normalization of filterbank weight functions, peak at 1.0 (HTK).
LOGMEL_OPT_FBNORM_AREA	1<<2	Perform area normalization of filterbank weight functions (Auditory).
MFCC_OPT_REMOVE_DC_MEAN	0<<3	For every STFT frame, evaluate and subtract the DC mean (HTK).
MFCC_OPT_DONT_REMOVE_DC_MEAN	1<<3	Do not remove DC mean (Auditory).
MFCC_OPT_PREEMPH_FRAMEBYFRAME	0<<4	Pre-emphasis filter state is reset between STFT frames (HTK).
MFCC_OPT_PREEMPH_CONTINUOUS	1<<4	Pre-emphasis filter state is reset once, during initialization (Auditory).
MFCC_OPT_DCT_NORMALIZED	0<<5	Multiply the DCT-II matrix by $\sqrt{2/N}$, where N is the transform size (HTK).

MFCC_OPT_DCT_ORTHOGONAL	1<<5	As above, and multiply the first row by $1/\sqrt{2}$ (Auditory).

STFT weighting window generator callback function

```
typedef void mfcc32x32_genWindow_cbfxn_t(void* host, fract32   * window, int len);
typedef void mfccf_genWindow_cbfxn_t    (void* host, float32_t * window, int len);
```

Type	Name	Size	Description
Input			
void*	host		User-supplied host handle
int	len		Length of the weighting window
Output			
fract32, float32_t	window	len	Generated weighting window samples; Q31 for 32x32; aligned by 16-bytes

Real-to-complex FFT callback function

```
typedef int  mfcc32x32_rfft_cbfxn_t( void * host, complex_fract32 * restrict y,
                                     fract32   * restrict x, int fftSize );
typedef void mfccf_rfft_cbfxn_t    ( void * host, complex_float   * restrict y,
                                     float32_t * restrict x, int fftSize );
```

Type	Name	Size	Description
Input			
void*	host		User-supplied host handle
int	fftSize		Transform size
fract32, float32_t	x	fftSize	Time-domain input signal
Output			
complex_fract32, complex_float	y	fftSize/2+1	Positive-frequency spectrum samples

32x32 variant of the function must return the block exponent of the Fourier image, which is the sum of bit shifts applied to data throughout the transform. Block exponent is a signed quantity, where positive values render shifting data to the right.

Notes:

1. Input and output arguments `x[fftSize]` and `y[fftSize/2+1]` do not overlap, and are aligned by 16-bytes.
2. The FFT function is allowed to re-use the input argument `x[fftSize]` for temporal storage of intermediate data.

Callbacks for fixed-point MFCC extractor (32x32).

```
typedef struct mfcc32x32_callback_tag
{
```

Type	Name	Description
void*	host	User-supplied host handle to be passed as the first input argument of a callback function.
mfcc32x32_genWindow_cbfxn_t *	genWindow	Optional STFT weighting window generator function. If NULL, then MFCC assumes the rectangular window, otherwise it invokes the user-supplied function during initialization.
mfcc32x32_rfft_cbfxn_t*	rfft	Real-to-complex FFT function, is called once per STFT sliding window update.

```
} mfcc32x32_callback_t;
```

Callbacks for floating-point MFCC extractor (single precision).

```
typedef struct mfccf_callback_tag
{
```

Type	Name	Description
void*	host	User-supplied host handle to be passed as the first input argument of a callback function.
mfccf_genWindow_cbfxn_t *	genWindow	Optional STFT weighting window generator

		function. If NULL, then MFCC assumes the rectangular window, otherwise it invokes the user-supplied function during initialization.
mfccf_rfft_cbfxn_t*	rfft	Real-to-complex FFT function, is called once per STFT sliding window update.

```
} mfccf_callback_t;
```

Fill the parameters structure with default values

```
void mfcc_getDefaultParams( mfcc_params_t * params );
```

Type	Name	Description
Output		
mfcc_params_t	params	Default parameters of MFCC features extraction.

Object allocation

```
size_t mfcc32x32_alloc( const mfcc_params_t * params );
size_t mfccf_alloc    ( const mfcc_params_t * params );
```

Type	Name	Description
Input		
mfcc_params_t	params	Parameters of MFCC features extraction.

Returns: size of memory to be allocated for an instance object, in bytes, or zero if parameters do not satisfy the restrictions (see below).

Object initialization

```
mfcc32x32_handle_t mfcc32x32_init( void * objmem, const mfcc_params_t * params,
                                   const mfcc32x32_callback_t * callback );
mfccf_handle_t     mfccf_init    ( void * objmem, const mfcc_params_t * params,
                                   const mfccf_callback_t * callback );
```

Type	Name	Description
Input		
void *	objmem	Memory block allocated for the instance object
mfcc_params_t	params	Parameters of MFCC features extraction.
mfcc32x32_callback_t, mfccf_callback_t	callback	User-supplied callback functions

Returns: handle to the object, or NULL if initialization failed.

Perform the sliding window STFT analysis and extract the MFCC features

```
void mfcc32x32_process( mfcc32x32_handle_t handle, void * restrict pScr,
                        fract32 * restrict cepstra,
                        const fract32 * restrict speech );
void mfccf_process    ( mfccf_handle_t handle, void * restrict pScr,
                        float32_t * restrict cepstra,
                        const float32_t * restrict speech );
```

Type	Name	Size	Description
Input			
fract32, float32_t	speech	stftHopLen	Speech samples; Q31 for 32x32.
Output			
fract32, float32_t	cepstra	cepstraNum	Cepstral coefficients; the number of fractional bits for 32x32 is defined by MFCC_CEPSTRA_FRACT_BITS macro constant.
Temporary			
void*	pScr		Scratch memory area for the processing function. To determine the scratch area size, use the respective helper function: mfcc<32x32|f>_getScratchSize().

2.11.3 HTK Delta Coefficients

Description

Function computes first order regression coefficients for HTK algorithm, see

http://www1.icsi.berkeley.edu/Speech/docs/HTKBook/node65_mn.html

For a set of `2*N+1` frames (each of `M` static coefficients) specified via the input argument `c[(2*N+1)*M]`, functions compute `M` delta coefficients and store results to the output argument `d[M]`.

The user may specify any window size `N`, but the computation is most efficient when `N` equals 2.

Precision

2 variants available:

Type	Description
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
f	floating point. Requires VFPU/SFPU core option

Algorithm

$$d_m = \frac{\sum_{n=1}^{N} n \cdot \left(c_{m+(N+n)\cdot M} - c_{m+(N-n)\cdot M}\right)}{2 \cdot \sum_{n=1}^{N} n^2}, m = \overline{0..M-1}$$

Prototype

```
void htkdelta32x32 (int32_t*   d,  const int32_t   * c, int M, int N);
void htkdeltaf     (float32_t* d,  const float32_t * c, int M, int N);
```

Type	Name	Size	Description
Input			
`int32_t`, `float32_t`	`c`	`(2*N+1)*M`	static coefficient frames. Coefficients that belong to the first frame are stored to `c[0..M-1]`, coefficients for the second frame - to `c[M..2*M-1]` and so forth.
`int`	`N`		window size
`int`	`M`		frame size
Output			
`int32_t`, `float32_t`	`d`	`M`	delta coefficients. The number of fractional positions for the fixed-point variant is the same as for the input argument `c[(2*N+1)*M]`

Returns: none

Restrictions

`c,d`	should not overlap
`c,d`	aligned on a 16-bytes boundary
`M`	12...30

2.12 Image Processing

This set of routines allows to process image data. Most routines support three formats:

- 8-bit unsigned grayscale data (suffix _gu8 in the API)
- 8-bit signed grayscale data (suffix _gs8 in the API)
- 16-bit signed grayscale data (suffix _gs16 in the API)

For 16-bit format all negative values are saturated to zero.

All images are represented by width, height and stride between successive rows. Pixel order is from left to right and from upside to downside. For performance reasons, all functions are divided into 2 categories:

- generic functions which process images with arbitrary parameters
- 'fast' functions which assume that the pointer for each row is aligned on 16-byte boundary. This also means that the pointer to the image is aligned and stride is a multiple of 8 for 8-bit images and multiple of 4 for 16-bit images.

As a general rule, all image transforms do not touch pixels outside output image area defined by width and height which means that output would be directly mapped existing image of bigger size.

```
typedef struct {
```

Type	Name	Description
`unsigned int`	`width`	width of image
`unsigned int`	`height`	height of image
`unsigned int`	`stride`	stride between consecutive rows, in pixels. Should be not less than width

```
} imgsize_ t;
```

2.12.1 Image Padding/Cropping

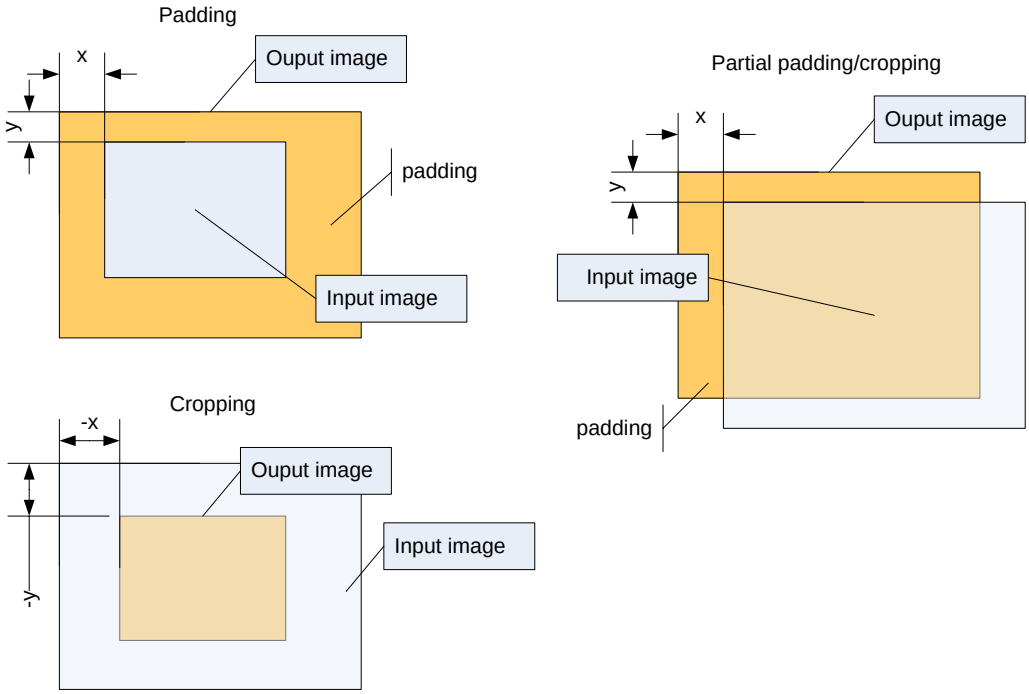
Description

Padding/cropping functions make left/right/top/bottom padding for input image. They copy input image to the output at specific position and fills borders either with specified color or by duplication of pixels from corresponding edges.
Region to be copies is set by position of left/upper corner of input in the coordinates of output image. If this region is outside the boundaries of output image, then function will crop required region from the input with padding of unused regions

Image formats

3 variants available:

Type	Description
gu8	8-bit unsigned grayscale data
gs8	8-bit signed grayscale data
gs16	16-bit signed grayscale data



Prototypes

```
void imgpad_gu8      ( void * pScr, void * outImg, const void * inImg,
                       const imgpad_params_t* params );
void imgpad_gs8    ( void * pScr, void * outImg, const void * inImg,
                       const imgpad_params_t* params );
void imgpad_gs16    ( void * pScr, void * outImg, const void * inImg,
                       const imgpad_params_t* params );
void imgfastpad_gs8( void * pScr, void * outImg, const void * inImg,
                       const imgpad_params_t* params );
void imgfastpad_gu8  ( void * pScr, void * outImg, const void * inImg,
                       const imgpad_params_t* params );
void imgfastpad_gs16( void * pScr, void * outImg, const void * inImg,
                       const imgpad_params_t* params );

typedef struct
{
```

Type	Name	Description
`imgsize_t`	`in`	input image size
`imgsize_t`	`out`	output image size
`int`	`x,y`	position of left upper corner of input image on the output if positions of input image corners are outside the output image, function will crop required region from the input
`int`	`fill`	fill color or special constant IMGPAG_EDGE to use edge pixels

```
} imgpad_params_t;
```

Arguments

Type	Name	Size	Description
Input			
	`inImg`		input image data
`imgpad_params_t`	`params`	1	parameters
Output			
	`outImg`		output mage data
Temporary			
	`pScr`		scratch memory, size is requested by corresponding `xxx_getScrtactSize` () function

Returned value none

Restrictions see general definitions in 2.12
Input and output image buffers can not overlapping

2.12.2 Image Histogram

Description Functions compute the histogram of input image. First bin corresponds minimum intensity (0 for signed 16-bit and unsigned 8-bit data, -128 for signed 8-bit) and the last bin corresponds to the maximum intensity (32767 for signed 16-bit , 255 for unsigned 8-bit data, 127 for signed 8-bit data)

Image formats 3 variants available:

Type	Description
gu8	8-bit unsigned grayscale data
gs8	8-bit signed grayscale data
gs16	16-bit signed grayscale data

Prototypes

```
void imghist_gu8       ( imghist_t * res,
                         const void * inImg, const imgsize_t* sz, int M);

void imgfasthist_gu8   ( imghist_t * res,
                         const void * inImg, const imgsize_t* sz, int M);

void imghist_gs8     ( imghist_t * res,
                         const void * inImg, const imgsize_t* sz, int M);

void imgfasthist_gs8 ( imghist_t * res,
                         const void * inImg, const imgsize_t* sz, int M);

void imghist_gs16     ( imghist_t * res,
                         const void * inImg, const imgsize_t* sz, int M);

void imgfasthist_gs16 ( imghist_t * res,
                         const void * inImg, const imgsize_t* sz, int M);
```

```
typedef struct {
```

Type	Name	Description
int32_t	mean	mean, Q15.16
int32_t	var	variance. Q31.0 (integral with no fractional bits)
int	M	number of bins
int32_t*	h	histogram values

```
} imghist_t;
```

Arguments

Type	Name	Size	Description
Input			
	inImg		input image data
imgsize_t	sz	1	image size
int32_t*	res->h	M	pointer to the array of histogram values
int	M		number of bins
Output			
imghist_t	res	1	resulted statistics

Returned value none

Restrictions see general definitions in 2.12

2.12.3 Image Normalization

Description Function normalize the intensity of pixels to the given range.
Linear normalization takes min/max values from the image and linearly normalizes them to the given range.
Nonlinear method maps input pixel values to the output range using tabulated transform with interpolation

Image formats

3 variants available:

Type	Description
gu8	8-bit unsigned grayscale data
gs8	8-bit signed grayscale data
gs16	16-bit signed grayscale data

Prototypes

Linear method

```
void imgnorm_gu8      ( void * outImg, const void * inImg, const imgsize_t* sz,
                        int minInt, int maxInt);

void imgfastnorm_gu8  ( void * outImg, const void * inImg, const imgsize_t* sz,
                        int minInt, int maxInt);

void imgnorm_gs8     ( void * outImg, const void * inImg, const imgsize_t* sz,
                        int minInt, int maxInt);

void imgfastnorm_gs 8( void * outImg, const void * inImg, const imgsize_t* sz,
                        int minInt, int maxInt);

void imgnorm_gs16    ( void * outImg, const void * inImg, const imgsize_t* sz,
                        int minInt, int maxInt);

void imgfastnorm_gs16( void * outImg, const void * inImg, const imgsize_t* sz,
                        int minInt, int maxInt);
```

Non-linear method

```
void imgnorm_gu8_nonlinear       ( void * outImg, const void * inImg,
                                   const imgsize_t* sz, const int16_t * tbl);

void imgfastnorm_gu8_nonlinear   ( void * outImg, const void * inImg,
                                   const imgsize_t* sz, const int16_t * tbl);

void imgnorm_gs8_nonlinear     ( void * outImg, const void * inImg,
                                   const imgsize_t* sz, const int16_t * tbl);

void imgfastnorm_gs8_nonlinear ( void * outImg, const void * inImg,
                                    const imgsize_t* sz, const int16_t * tbl);

void imgnorm_gs16_nonlinear     ( void * outImg, const void * inImg,
                                   const imgsize_t* sz, const int16_t * tbl);

void imgfastnorm_gs16_nonlinear ( void * outImg, const void * inImg,
                                   const imgsize_t* sz, const int16_t * tbl);
```

Arguments

Type	Name	Size	Description
Input			
	inImg		input image data
imgsize_t	sz	1	image size
int	minInt, maxInt		output range (for linear method)
int16_t	tbl	64	64-points tabulated transform (for nonlinear method), see example in 4.3
Output			
	outImg		output image data

Returned value

none

Restrictions

see general definitions in 2.12
Input and output image buffers can not overlapping

2.12.4 RGB/YUV Conversion

Description

Functions make conversion between RGB and YUV formats. RGB/YUV data go in separate planes

Image formats

2 variants available:

Type	Description

	8-bit unsigned data
16	16-bit signed data

Algorithm

RGB->YUV

$$Y = c_0 R + c_1 G + c_2 B$$

$$U = c_3 R + c_4 G + c_5 B + bias$$

$$V = c_6 R + c_7 G + c_8 B + bias$$

YUV->RGB

$$R = Y + c_9 (V - bias)$$

$$G = Y + c_{10} (U - bias) + c_{11} (V - bias)$$

$$B = Y + c_{12} (U - bias)$$

where `bias` is 128 for 8-bit data, 16384 for 16-bit data
Transform coefficients are selected to fit appropriate encoding standard

Prototypes

From RGB to YUV:

```
void imgconvert_rgbyuv( void * outImgY,
                        void * outImgU,
                        void * outImgV,
                        const void * inImgR,
                        const void * inImgG,
                        const void * inImgB,
                        const int32_t *c,
                        const imgsize_t* sz);

void imgfastconvert_rgbyuv
                      ( void * outImgY,
                        void * outImgU,
                        void * outImgV,
                        const void * inImgR,
                        const void * inImgG,
                        const void * inImgB,
                        const int32_t *c,
                        const imgsize_t* sz);


void imgconvert_rgbyuv16( void * outImgY,
                        void * outImgU,
                        void * outImgV,
                        const void * inImgR,
                        const void * inImgG,
                        const void * inImgB,
                        const int32_t *c,
                        const imgsize_t* sz);

void imgfastconvert_rgbyuv16
                      ( void * outImgY,
                        void * outImgU,
                        void * outImgV,
                        const void * inImgR,
                        const void * inImgG,
                        const void * inImgB,
                        const int32_t *c,
                        const imgsize_t* sz);
```

Arguments

Type	Name	Size	Description
Input			
	inImgR, inImgG, inImgB		input image data for RGB color planes
int32_t	c	13	transform coefficients, Q29
imgsize_t	sz	1	image size
Output			
	outImgY, outImgU, outImgV		output image data for YUV color planes

Prototypes

From YUV to RGB:

```
void imgconvert_yuvrgb( void * outImgR,
                        void * outImgG,
                        void * outImgB,
                        const void * inImgY,
                        const void * inImgU,
                        const void * inImgV,
                        const int32_t *c,
                        const imgsize_t* sz);

void imgfastconvert_yuvrgb
                      ( void * outImgR,
                        void * outImgG,
                        void * outImgB,
                        const void * inImgY,
                        const void * inImgU,
                        const void * inImgV,
                        const int32_t *c,
                        const imgsize_t* sz);

 void imgconvert_yuvrgb16
                      ( void * outImgR,
                        void * outImgG,
                        void * outImgB,
                        const void * inImgY,
                        const void * inImgU,
                        const void * inImgV,
                        const int32_t *c,
                        const imgsize_t* sz);

void imgfastconvert_yuvrgb16
                      ( void * outImgR,
                        void * outImgG,
                        void * outImgB,
                        const void * inImgY,
                        const void * inImgU,
                        const void * inImgV,
                        const int32_t *c,
                        const imgsize_t* sz);
```

Arguments

Type	Name	Size	Description
Input			
	inImgY, inImgU, inImgV		input image data for YUV color planes
int32_t	c	13	transform coefficients, Q29
imgsize_t	sz	1	image size
Output			
	outImgR, outImgG, outImgB		output image data for RGB color planes

Returned value

none

Restrictions

see general definitions in 2.12
Input and output image buffers can not overlapping

2.12.5 Image Interleave/Deinterleave

Description

Functions make conversion between packed and planar image representation. In the packed representation pixels from different color planes are interleaved so they are grouped in 24/48-bit chunks. It is assumed that the width and the stride of packed image are 3 times bigger than for input image

Image formats

2 variants available:

Type	Description
	8-bit signed or unsigned data
16	16-bit signed data

Prototypes

From planar to packed:

```
void imginterleave    (      void * restrict outImg,
                       const void * restrict inImgR,
                       const void * restrict inImgG,
                       const void * restrict inImgB,
                       const imgsize_t* sz);

void imgfastinterleave(      void * restrict outImg,
                       const void * restrict inImgR,
                       const void * restrict inImgG,
                       const void * restrict inImgB,
                       const imgsize_t* sz);

void imginterleave16  (      void * restrict outImg,
                       const void * restrict inImgR,
                       const void * restrict inImgG,
                       const void * restrict inImgB,
                       const imgsize_t* sz);

void imgfastinterleave16 (   void * restrict outImg,
                       const void * restrict inImgR,
                       const void * restrict inImgG,
                       const void * restrict inImgB,
                       const imgsize_t* sz);
```

Arguments

Type	Name	Size	Description
Input			
	inImgR, inImgG, inImgB		input image data for RGB color planes
imgsize_t	sz	1	input image size
Output			
	outImg		output packed image data

Prototypes

From packed to planar:

```
void imgdeinterleave  (       void * outImgR,
                              void * outImgG,
                              void * outImgB,
                        const void * inImg,
                        const imgsize_t* sz);

void imgfastdeinterleave(     void * outImgR,
                              void * outImgG,
                              void * outImgB,
                        const void * restrict inImg,
                        const imgsize_t* sz);

void imgdeinterleave16(       void * outImgR,
                              void * outImgG,
                              void * outImgB,
                        const void * inImg,
                        const imgsize_t* sz);

void imgfastdeinterleave16(   void * outImgR,
                              void * outImgG,
                              void * outImgB,
```

```
                                    const void * restrict inImg,
                                    const imgsize_t* sz);
```

Arguments

Type	Name	Size	Description
Input			
	inImg		input packed image data
imgsize_t	sz	1	output image size
Output			
	outImgR, outImgG, outImgB		output image data for RGB color planes

Returned value

none

Restrictions

see general definitions in 2.12
Input and output image buffers can not overlapping

2.12.6 Image Resize

Description

These functions resize images. There are 3 methods for transformation: neighbor, bilinear and bicubic. Resize process is divided onto initialization and processing phases. On the initialization, function returns the size of memory requesting for persistent data (i.e. coefficients) needed for downsampling and upsampling operations. In the processing phase, input image is converted to the output image according to the transformation parameters. Additionally, image resize requires some amount of temporary (scratch) data which may not be kept between successive calls of processing function and might be reused.

Image formats

3 variants available:

Type	Description
gu8	8-bit unsigned grayscale data
gs8	8-bit signed grayscale data
gs16	16-bit signed grayscale data

Prototypes

```
typedef struct {
```

Type	Name	Description
imgsize_t	in	size of input image
imgsize_t	out	size of output image
imgresize_method_t	method	resize method: one of 3 possible constants: imgresize_method_nearest imgresize_method_bilinear imgresize_method_bicubic

```
} imgresize_params_t;
```

Allocation functions:

```
size_t imgresize_gu8_alloc      ( const imgresize_params_t * params );
size_t imgresize_gs8_alloc    ( const imgresize_params_t * params );
size_t imgresize_gs16_alloc    ( const imgresize_params_t * params );
size_t imgfastresize_gu8_alloc  ( const imgresize_params_t * params );
size_t imgfastresize_gs8_alloc( const imgresize_params_t * params );
size_t imgfastresize_gs16_alloc( const imgresize_params_t * params );
```

Initialization functions:

```
imgresize_handle_t imgresize_gu8_init
    ( void * objmem, const imgresize_params_t * params );
imgresize_handle_t imgresize_gs8_init
    ( void * objmem, const imgresize_params_t * params );
imgresize_handle_t imgresize_gs16_init
    ( void * objmem, const imgresize_params_t * params );
imgresize_handle_t imgfastresize_gu8_init
    ( void * objmem, const imgresize_params_t * params );
imgresize_handle_t imgfastresize_gs8_init
    ( void * objmem, const imgresize_params_t * params );
imgresize_handle_t imgfastresize_gs16_init
    ( void * objmem, const imgresize_params_t * params );
```

Processing functions:

```
void imgresize_gu8_process      ( imgresize_handle_t handle,
                          void * pScr, void * outImg, const void * inImg);
void imgresize_gs8_process    ( imgresize_handle_t handle,
                          void * pScr, void * outImg, const void * inImg);
void imgresize_gs16_process    ( imgresize_handle_t handle,
                          void * pScr, void * outImg, const void * inImg);
void imgfastresize_gu8_process  ( imgresize_handle_t handle,
                          void * pScr, void * outImg, const void * inImg);
void imgfastresize_gs8_process( imgresize_handle_t handle,
                          void * pScr, void * outImg, const void * inImg);
void imgfastresize_gs16_process( imgresize_handle_t handle,
                          void * pScr, void * outImg, const void * inImg);
```

requesting the scratch size

```
size_t imgresize_gu8_getScratchSize      ( const imgresize_params_t * params );
size_t imgresize_gs8_getScratchSize    ( const imgresize_params_t * params );
size_t imgresize_gs16_getScratchSize    ( const imgresize_params_t * params );
size_t imgfastresize_gu8_getScratchSize  ( const imgresize_params_t * params );
size_t imgfastresize_gs8_getScratchSize( const imgresize_params_t * params );
size_t imgfastresize_gs16_getScratchSize( const imgresize_params_t * params );
```

Arguments

Type	Name	Size	Description
Input			
	`inImg`		input image data
`imgresize_params_t`	`params`	1	input image size
Output			
	`outImg`		output image data

Returned value

none

Restrictions

see general definitions in 2.12
Input and output image buffers can not overlapping

2.12.7 Image Rotation

Description

Image rotation functions rotate input image counter clockwise about their center. They use:

- dedicated variants for 90, 180 and -90 degree rotations
- 3-shear steps for small angles in range -pi/4...pi/4
- proper combinations of above variants for other angles

Since all of these algorithms show very different computational speed, overall transformation performance depends not only from the image size, but from the rotation angle as well.
Except for the rotation on 90, 180 and -90 degrees, the size of resulted image depends on the rotation angle and all unused portions of image will be filled with the fill color. Required output size should be requested by `xxx_getOutSize()` API, however user may change the stride of output image to make possible direct mapping to existing image framebuffer.
Rotation API consists of 5 categories:

- allocation function returning amount of memory for persistent data (i.e. interpolation arrays) required for rotation of image of given size to the given angle
- initialization function which compute and fill all persistent data
- requesting the scratch size required for transform
- requesting the size of output image
- processing function for rotation a single image

Note: if user processes images of the same size and with the same rotation angle, the next allocation, initialization functions might be omitted and only processing function will be called

Image formats

3 variants available:

Type	Description
gu8	8-bit unsigned grayscale data
gs8	8-bit signed grayscale data

gs16	16-bit signed grayscale data

Prototypes

```
typedef struct {
```

Type	Name	Description
imgsize_t	in	size of input image
int16_t	angleQ15	angle scaled by pi, counterclockwise, in Q15, (so 90 degrees counter clockwise corresponds to 0x4000, 180 degrees - 0x8000, 270 degrees -0xc000)
int	fill	fill color for unused image parts

```
} imgrotate_params_t;
```

Allocation functions:

```
size_t imgrotate_gu8_alloc      ( const imgrotate_params_t * params );
size_t imgfastrotate_gu8_alloc  ( const imgrotate_params_t * params );
size_t imgrotate_gs8_alloc    ( const imgrotate_params_t * params );
size_t imgfastrotate_gs8_alloc( const imgrotate_params_t * params );
size_t imgrotate_gs16_alloc    ( const imgrotate_params_t * params );
size_t imgfastrotate_gs16_alloc( const imgrotate_params_t * params );
```

Initialization functions:

```
imgrotate_handle_t imgrotate_gu8_init       (void * objmem,
                                             const imgrotate_params_t * params);
imgrotate_handle_t imgfastrotate_gu8_init   (void * objmem,
                                             const imgrotate_params_t * params);
imgrotate_handle_t imgrotate_gs8_init     (void * objmem,
                                             const imgrotate_params_t * params);
imgrotate_handle_t imgfastrotate_gs8_init (void * objmem,
                                             const imgrotate_params_t * params);
imgrotate_handle_t imgrotate_gs16_init   (void * objmem,
                                             const imgrotate_params_t * params );
imgrotate_handle_t imgfastrotate_gs16_init(void * objmem,
                                              const imgrotate_params_t * params);
```

Request output image size:

```
void imgrotate_gu8_getOutSize      (imgsize_t * outSz,
                                    const imgrotate_params_t * params);
void imgfastrotate_gu8_getOutSize  (imgsize_t * outSz,
                                    const imgrotate_params_t * params);
void imgrotate_gs8_getOutSize    (imgsize_t * outSz,
                                    const imgrotate_params_t * params);
void imgfastrotate_gs8_getOutSize(imgsize_t * outSz,
                                    const imgrotate_params_t * params);
void imgrotate_gs16_getOutSize    (imgsize_t * outSz,
                                    const imgrotate_params_t * params);
void imgfastrotate_gs16_getOutSize(imgsize_t * outSz,
                                    const imgrotate_params_t * params);
```

Processing functions:

```
void imgrotate_gu8_process      ( imgrotate_handle_t handle, void * pScr,
               void * outImg, const void * inImg, const imgsize_t * outSz);
void imgfastrotate_gu8_process  ( imgrotate_handle_t handle, void * pScr,
               void * outImg, const void * inImg, const imgsize_t * outSz);
void imgrotate_gs8_process    ( imgrotate_handle_t handle, void * pScr,
               void * outImg, const void * inImg, const imgsize_t * outSz);
void imgfastrotate_gs8_process( imgrotate_handle_t handle, void * pScr,
               void * outImg, const void * inImg, const imgsize_t * outSz);
void imgrotate_gs16_process    ( imgrotate_handle_t handle, void * pScr,
               void * outImg, const void * inImg, const imgsize_t * outSz);
void imgfastrotate_gs16_process( imgrotate_handle_t handle, void * pScr,
               void * outImg, const void * inImg, const imgsize_t * outSz);
```

Requesting the scratch size:

```
size_t imgrotate_gu8_getScratchSize      ( const imgrotate_params_t * params );
size_t imgfastrotate_gu8_getScratchSize  ( const imgrotate_params_t * params );
size_t imgrotate_gs8_getScratchSize    ( const imgrotate_params_t * params );
size_t imgfastrotate_gs8_getScratchSize( const imgrotate_params_t * params );
size_t imgrotate_gs16_getScratchSize    ( const imgrotate_params_t * params );
```

```
size_t imgfastrotate_gs16_getScratchSize( const imgrotate_params_t * params );
```

Arguments

Type	Name	Size	Description
Input			
	inImg		input image data
imgresize_params_t	params	1	input image size
imgsize_t	outSz	1	width, height and stride of output image. NOTE: width and height should be set by xxx_getOutSize() API, but the stride might be changed by user
Output			
	outImg		output image data

Returned value

none

Restrictions

see general definitions in 2.12
Input and output image buffers can not overlapping

2.12.8 Image 2D FFT

Description

Image 2D-FFT makes special kind of Fourier transform over image data. It supports a number of resolutions. For processing of arbitrary sizes, user might pad original data to the closest available dimension.

Important notes:

1. Accuracy of the fixed point 2D FFT suffers from the presence of unknown overall image brightness so for the optimal results, mean pixel value should be precomputed (i.e. by histogram function) and passed on the input of FFT.
2. FFT processes the data with autoscaling similarly as it is done in regular FFT routines so it returns the scaled transform and number of right shifts done during scaling procedure. Inverse FFT uses this scale factor to recover original data range and saturates results to 8-bit unsigned/signed or 16-bit signed data.
3. IFFT ignores zero-th tap of transform. Instead, it uses dedicated parameter to set mean intensity of the output image
4. FFT accepts input in desired format and form output in form of 2D array of complex 16-bit numbers
5. IFFT makes in-place transform so input spectrum will be damaged after the call

Supported dimensions:
64, 96, 128, 144, 176, 240, 256, 288, 320, 352, 384, 480, 512, 576, 640

Precision

3 variants available:

Type	Description
gu8	8-bit unsigned grayscale data
gs8	8-bit signed grayscale data
gs16	16-bit signed grayscale data

Prototype (FFT)

```
int imgfft_gu8 (void *pScr,
                complex_fract16* y, const void* img,
                int32_t mean, const imgsize_t* sz);
int imgfft_gs8 (void *pScr,
                complex_fract16* y, const void* img,
                int32_t mean, const imgsize_t* sz);
int imgfft_gs16(void *pScr,
                complex_fract16* y, const void* img,
                int32_t mean, const imgsize_t* sz);
```

Request the scratch size:

```
size_t imgfft_gu8_getScratchSize   (const imgsize_t* sz);
size_t imgfft_gs8_getScratchSize (const imgsize_t* sz);
size_t imgfft_gs16_getScratchSize (const imgsize_t* sz);
```

Arguments

Type	Name	Size	Description
Input			
uint8_t, int16_t	img		input image
imgsize_t	sz	1	size, see list of supported dimensions
int32_t	mean		mean value (average intensity) over the image, Q15.16
Output			
complex_fract16	y	2*w*h	output spectrum. Real and imaginary data are interleaved and real data goes first. w and h - width and height of original image

Returned value

total number of right shifts occurred during scaling procedure

Prototype (IFFT)

Inverse transform:

```
void imgifft_gu8 (void *pScr,
                  void* img, complex_fract16* y,
                  int32_t mean, const imgsize_t* sz,
                  int fftscale);
void imgifft_gs8 (void *pScr,
                  void* img, complex_fract16* y,
                  int32_t mean, const imgsize_t* sz,
                  int fftscale);
void imgifft_gs16(void *pScr,
                  void* img, complex_fract16* y,
                  int32_t mean, const imgsize_t* sz,
                  int fftscale);
```

Request the scratch size:

```
size_t imgifft_gu8_getScratchSize   (const imgsize_t* sz);
size_t imgifft_gs8_getScratchSize (const imgsize_t* sz);
size_t imgifft_gs16_getScratchSize (const imgsize_t* sz);
```

Arguments

Type	Name	Size	Description
Input			
complex_fract16	y	2*w*h	input spectrum. w and h - width and height of output image
int32_t	mean		desired mean value (average intensity) over the output image, Q15.16
int	fftscale		scale factor returned by FFT
imgsize_t	sz	1	output image size
Output			
uint8_t, int16_t	img		output image

Returned value

none

Restrictions

img,y should be aligned on a 16-bytes boundary
image dimensions should be selected from the list of supported dimensions
image stride should be a multiple of 8 for 8-bit images and 4 for 16-bit images

2.13 Identification Routines

2.13.1 Library Version Request

Description

This function returns library version information.

Prototype

```
void NatureDSP_Signal_get_library_version(char *version_string);
```

Arguments

Type	Name	Size	Description
Output			
`char`	`version_string`	`>=30`	buffer to store version information

Returned value None

Restrictions version_string must points to a buffer large enough to hold up to 30 characters

Conditions for optimum performance: None

2.13.2 Library API Version Request

Description This function returns library API version information.

Prototype

```
void NatureDSP_Signal_get_library_api_version(char *version_string);
```

Arguments

Type	Name	Size	Description
Output			
`char`	`version_string`	`>=30`	buffer to store version information

Returned value None

Restrictions version_string must points to a buffer large enough to hold up to 30 characters

2.13.3 Library API Capability Request

Description This function returns non-zero if given function (by its address) is supported by specific processor capabilities (i.e. VFPU/SFPU option).

NOTE:

1. in the gcc/xcc environment, calls of this function are not necessary - if function pointer is non-zero it means it is supported. VisualStudio linker does not support section removal so this function might be used for running library under MSVC environment
2. Very few library functions may disable their capabilities dynamically (only for particular combination of input parameters). Behavior for such situation is defined in the description of those functions.

Prototype

```
int NatureDSP_Signal_isPresent(NatureDSP_Signal_funptr fun)
```

Arguments

Type	Name	Description
Input:		
`NatureDSP_Signal_funptr`	`fun`	one of library functions

Returned Value non-zero if function is supported by library

3 Test Environment and Examples

3.1 Supported Use Environment, Configurations and Targets

NatureDSP Signal library and corresponding testdriver is supported to be built and test using Xtensa Xplorer IDE running under Windows, or Linux operating system.

Library is compatible with HiFi cores having following options:

- HiFi5 base ISA
- HiFi5 Vector/Scalar FP
- NSA/NSAU ISA option
- MIN/MAX ISA option
- Boolean Registers ISA option
- Little endian target

3.2 Building the NatureDSP Signal Library and the Testdriver

3.2.1 Importing the workspaces in Xtensa Xplorer

NatureDSP Signal Library for HiFi5 VFPU is provided as two workspaces:

- Library workspace `hifi5_library.xws`

  This workspace contains optimized kernels, modules as required for demo workspaces

- Demo workspace `hifi5_demo.xws`

  This contains the `hifi5_demo` demo project.

Import these two workspaces (`.xws`) in Xtensa Xplorer (XX) as “Xtensa Xplorer workspace”.

Make sure that the library workspace is imported first. This is because the project in the demo workspace has a dependency on the library projects, and the dependency is not correctly set if the library projects are not present when the demo workspace is imported.

3.2.2 Building and Running Tests

To build the library: In XX, select the desired library project to build, and Debug or Release target, and build.

To build the test bench: In XX, select the `hifi5_demo` project, select Debug or Release target, and build.

To run the test bench, select `hifi5_demo` project, and Run. This will execute each routine in the `hifi5_library` in cycles performance (MIPS) mode.

For performance measurement with memory modeling, build the test bench with `MEM_MODEL=1` (set in build properties) and use `--mem_model` as runtime argument for execution.

Use `--turbo` as runtime argument to test library for functional correctness

3.2.3 Command-line Options

You may wish to launch a separate test by passing command-line options to the executable:
Running the Testdriver without options performs functional testing of library. Additionally, it may collect statistics and generate validation report showing the number of calls of each specific library function, amount of data passed to/from, sorts of specific tests performed, etc.
Running performance tests for all library functions or for specific category is controlled by command line option `-mips`. In that case, functional testing is not performed and validation report will be empty.
Brief performance data are formed with `-mips -verbose`. Detailed performance data are prepared with `-mips -full`.

You may wish to launch a separate test by passing command-line options to the executable:

Package	API	Meaning	Option
		List of available options	`-help or -h`
		Performance test	`-mips`
		Functional tests	`-func`
		Generate validation report and statistics after completion of functional testing	`-vreport`
		test fixed point functions only	`-phase1`
		test floating point functions only	`-phase2`
		For functional tests, this switch instructs to use bigger data vectors from directory `vectors_full` instead of `vectors_brief` (test time might be 3 to 5 times longer). For performance test, it controls amount of tests performed for each library function. With this switch the test tool forms detailed testing using bigger set of function parametes, if not - it just makes brief performance data.	`-full`
		This switch controls amount of tests executed (in contrary with `-full`)	`-brief`
		Verbose reporing. For functional tests, it controls verbosilty of test reports (i.e. shows number, type of tests performed and detailed error statistics if some test is failed). For performance tests, it adds textual description of each function under the test to the performance log.	`-verbose`
FIR Filters and Related Functions	2.1	all FIR filters	**`-fir`**
		filtering	`-firblk`
		decimation	`-firdec`
		interpolation	`-firint`
		correlation, convolution, dispreading, LMS, polyphase	`-firother`
		2D-convolution	`-conv2d`
IIR Filters	2.1.14	all IIR filters	**`-iir`**
		biquad filters	`-iirbq`
		lattice filters	`-iirlt`
		kalman updates	`-iirkal`
Math Functions	2.3	all math functions	**`-math`**
		vectorized math	`-mathv`
		vectorized math (fast variants)	`-mathvf`
		scalar math	`-maths`
Complex Math Functions	2.4	all complex functions	**`-complex`**
		vectorized complex math	`-complexv`
		scalar complex math	`-complexs`
Vector Operations	2.5	vector operations tests	**`-vector`**

Package	API	Meaning	Option
Emulated Floating Point Operations	2.6	emulated floating point tests	**-ef**
Matrix Operations	2.7	all matrix operations tests	**-matop**
Matrix Decomposition and Inversion Functions	2.8	all matrix decomposition and inversion	**-matinv**
FFT Routines	2.10	all FFT and DCT	**-fft**
		complex FFT	-cfft
		real FFT	-rfft
		mixed radix complex FFT	-cnfft
		mixed radix real FFT	-rnfft
		complex FFT with optimized memory	-cfftie
		real FFT with optimized memory	-rfftie
		power spectrum	-spectrum
		DCT	-dct
MFCC	2.11	MFCC feature extraction	-mfcc
Fitting and Interpolation Routines	2.9	all fitting tests	**-fit**
		polynomial fitting	-pfit

4 Appendix

4.1 Matlab Code for Conversion of SOS Matrix to Coefficients of IIR Functions

Below is example Matlab code to simplify conversion of SOS+G matrices given from the filter design tools into the format of IIR filtering functions.

4.1.1 bqriir16x16_df1, bqriir32x16_df1 conversion

```
%--------------------------------------------------
% convert SOS+G to coefficients of IIR filter
% (bqriir32x16_df1 function)
% parameters:
% SOS,G    - SOS matrix and gain vector G
% Fs       - sample rate
% nfft     - FFT length for analisys
% output:
% coef     - vector with coefficients, Q30
% gain     - biquad gains, Q15
% scale    - final scale factor (amount of left shifts)
%--------------------------------------------------
function [coef,gain,scale]=cvtsos_bqriir32x16_df1(SOS,G,Fs,nfft)
sz=size(SOS);
M=sz(1);
coef=[];
f=(0:nfft-1)/nfft*(Fs/2);
tf0=sos2freqz(SOS,G,nfft);

Gtotal=prod(G);
G=ones(1,M+1);

% define gain for each stage to provide maximim of tf for intermediate output <=0.5
for m=1:M
    tf=sos2freqz(SOS(1:m,:),[G(1:m) 1],nfft);
    tfmax=max(abs(tf));
    G(m)=min(1,0.5/tfmax);
end

% define last stage shift
dg=Gtotal/prod(G);
scale=ceil(log2(dg));
% correct coefficient of the last stage
d=pow2(1,scale)/dg;
G(M)=G(M)/d;
% output b,a
coef=SOS; coef(:,4)=[]; coef=reshape(coef.',1,numel(coef));
% and convert coefficients to given format
coef = int16(round(16384.*coef));
gain = int16(round(32768.*G(1:M)));
scale= int16(scale);
% check results and plot final filter response
sos=reshape(double(coef),5,M).'/16384;
sos=[sos(:,1:3) ones(M,1) sos(:,4:5)];
g=double(gain)/32768;
g(M+1)=pow2(1,double(scale));
tf=sos2freqz(sos,g,nfft);
plot(f,20*log10(abs(tf)),f,20*log10(abs(tf0))); ylim([-80 0]); title('transfer function, dB'); grid
on;

% convert SOS/G to frequency response
function [tf]=sos2freqz(SOS,G,nfft)
sz=size(SOS);
M=sz(1);
[b,a]=sos2tf(SOS(1,:),[G(1) G(end)]);
```

```
tf=freqz(b,a,nfft);
for m=2:M
    [b,a]=sos2tf(SOS(m,:),[G(m) 1]);
    tf=tf.*freqz(b,a,nfft);
end
```

4.1.2 bqriir16x16_df2, bqriir32x16_df2 conversion

```
%-------------------------------------------------
% convert SOS+G to coefficients of IIR filter
% (bqriir32x16_df2 function)
% parameters:
% SOS,G    - SOS matrix and gain vector G
% Fs       - sample rate
% nfft     - FFT length for analisys
% output:
% coef     - vector with coefficients, Q14
% gain     - biquad gains, Q15
% scale    - final scale factor (amount of left shifts)
%-------------------------------------------------
function [coef,gain,scale]=cvtsos_bqriir32x16_df2(SOS,G,Fs,nfft)

sz=size(SOS);
M=sz(1);
coef=[];
f=(0:nfft-1)/nfft*(Fs/2);
tf0=sos2freqz(SOS,G,nfft);
Gtotal=prod(G);
G=ones(1,M+1);

% define gain for each stage to provide maximim of tf for
% intermediate outputs <=0.5
% note: for DF2 structure we have to check 2 points:
% b0,b1,b2,a1,a2 and 1,0,0,a1,a2
for m=1:M
    tf=sos2freqz(SOS(1:m,:),[G(1:m) 1],nfft);
    tfmax0=max(abs(tf));
    tf=sos2freqz([SOS(1:m-1,:);1 0 0 1 SOS(m,5:6)] ,[G(1:m) 1],nfft);
    tfmax1=max(abs(tf));
    tfmax = max(tfmax0,tfmax1);
    G(m)=min(1,0.5/tfmax);
end

% define last stage shift
dg=Gtotal/prod(G);
scale=ceil(log2(dg));
% correct coefficient of the last stage
d=pow2(1,scale)/dg;
G(M)=G(M)/d;
% output b,a
coef=SOS; coef(:,4)=[]; coef=reshape(coef.',1,numel(coef));
% and convert coefficients to given format
coef = int16(round(16384.*coef));
gain = int16(round(32768.*G(1:M)));
scale= int16(scale);

% check results and plot final filter response
sos=reshape(double(coef),5,M).'/16384;
sos=[sos(:,1:3) ones(M,1) sos(:,4:5)];
g=double(gain)/32768;
g(M+1)=pow2(1,double(scale));
tf=sos2freqz(sos,g,nfft);
plot(f,20*log10(abs(tf)),f,20*log10(abs(tf0))); ylim([-80 0]); title('transfer function, dB'); grid
on;

% convert SOS/G to frequency response
function [tf]=sos2freqz(SOS,G,nfft)
sz=size(SOS);
M=sz(1);
[b,a]=sos2tf(SOS(1,:),[G(1) G(end)]);
tf=freqz(b,a,nfft);
for m=2:M
```

```
    [b,a]=sos2tf(SOS(m,:),[G(m) 1]);
    tf=tf.*freqz(b,a,nfft);
end
```

4.1.3 bqriir32x32_df1 conversion

```
%-----------------------------------------------------
% convert SOS+G to coefficients of IIR filter
% (bqriir32x32_df1 function)
% parameters:
% SOS,G    - SOS matrix and gain vector G
% Fs       - sample rate
% nfft     - FFT length for analisys
% output:
% coef     - vector with coefficients, Q30
% gain     - biquad gains, Q30
% scale    - final scale factor (amount of left shifts)
%-----------------------------------------------------
function [coef,gain,scale]=cvtsos_bqriir32x32_df1(SOS,G,Fs,nfft)

sz=size(SOS);
M=sz(1);
coef=[];
f=(0:nfft-1)/nfft*(Fs/2);
tf0=sos2freqz(SOS,G,nfft);
Gtotal=prod(G);
G=ones(1,M+1);

% define gain for each stage to provide maximim of tf for intermediate output <=0.5
for m=1:M
    tf=sos2freqz(SOS(1:m,:),[G(1:m) 1],nfft);
    tfmax=max(abs(tf));
    G(m)=min(1,0.5/tfmax);
end

% define last stage shift
dg=Gtotal/prod(G);
scale=ceil(log2(dg));
% correct coefficient of the last stage
d=pow2(1,scale)/dg;
G(M)=G(M)/d;
% output b,a
coef=SOS; coef(:,4)=[]; coef=reshape(coef.',1,numel(coef));
% and convert coefficients to given format
coef = int32(round(1073741824.*coef));
gain = int16(round(32768.*G(1:M)));
scale= int16(scale);
% check results and plot final filter response
sos=reshape(double(coef),5,M).'/1073741824;
sos=[sos(:,1:3) ones(M,1) sos(:,4:5)];
g=double(gain)/32768;
g(M+1)=pow2(1,double(scale));
tf=sos2freqz(sos,g,nfft);
plot(f,20*log10(abs(tf)),f,20*log10(abs(tf0))); ylim([-80 0]); title('transfer function, dB'); grid
on;

% convert SOS/G to frequency response
function [tf]=sos2freqz(SOS,G,nfft)
sz=size(SOS);
M=sz(1);
[b,a]=sos2tf(SOS(1,:),[G(1) G(end)]);
tf=freqz(b,a,nfft);
for m=2:M
    [b,a]=sos2tf(SOS(m,:),[G(m) 1]);
    tf=tf.*freqz(b,a,nfft);
end
```

4.1.4 bqriir32x32_df2 conversion

```
%-----------------------------------------------------
% convert SOS+G to coefficients of IIR filter
```

```
% (bqriir32x32_df2 function)
% parameters:
% SOS,G    - SOS matrix and gain vector G
% Fs       - sample rate
% nfft     - FFT length for analisys
% output:
% coef     - vector with coefficients, Q30
% gain     - biquad gains, Q15
% scale    - final scale factor (amount of left shifts)
%--------------------------------------------------
function [coef,gain,scale]=cvtsos_bqriir32x32_df2(SOS,G,Fs,nfft)

sz=size(SOS);
M=sz(1);
coef=[];
f=(0:nfft-1)/nfft*(Fs/2);
tf0=sos2freqz(SOS,G,nfft);
Gtotal=prod(G);
G=ones(1,M+1);

% define gain for each stage to provide maximim of tf for
% intermediate outputs <=0.5
% note: for DF2 structure we have to check 2 points:
% b0,b1,b2,a1,a2 and 1,0,0,a1,a2
for m=1:M
    tf=sos2freqz(SOS(1:m,:),[G(1:m) 1],nfft);
    tfmax0=max(abs(tf));
    tf=sos2freqz([SOS(1:m-1,:);1 0 0 1 SOS(m,5:6)] ,[G(1:m) 1],nfft);
    tfmax1=max(abs(tf));
    tfmax = max(tfmax0,tfmax1);
    G(m)=min(1,0.5/tfmax);
end

% define last stage shift
dg=Gtotal/prod(G);
scale=ceil(log2(dg));
% correct coefficient of the last stage
d=pow2(1,scale)/dg;
G(M)=G(M)/d;
% output b,a
coef=SOS; coef(:,4)=[]; coef=reshape(coef.',1,numel(coef));
% and convert coefficients to given format
coef = int32(round(1073741824.*coef));
gain = int16(round(32768.*G(1:M)));
scale= int16(scale);

% check results and plot final filter response
sos=reshape(double(coef),5,M).'/1073741824;
sos=[sos(:,1:3) ones(M,1) sos(:,4:5)];
g=double(gain)/32768;
g(M+1)=pow2(1,double(scale));
tf=sos2freqz(sos,g,nfft);
plot(f,20*log10(abs(tf)),f,20*log10(abs(tf0))); ylim([-80 0]); title('transfer function, dB'); grid
on;

% convert SOS/G to frequency response
function [tf]=sos2freqz(SOS,G,nfft)
sz=size(SOS);
M=sz(1);
[b,a]=sos2tf(SOS(1,:),[G(1) G(end)]);
tf=freqz(b,a,nfft);
for m=2:M
    [b,a]=sos2tf(SOS(m,:),[G(m) 1]);
    tf=tf.*freqz(b,a,nfft);
end
```

4.1.5 bqriirf_df1, bqriirf_df2, bqriirf_df2t conversion

```
%--------------------------------------------------
% convert SOS+G coefficients to the coefficients for
% iirdf1f,iirdf2f,iirdf2tf,ciirdf1f routines
% parameters:
```

```
% SOS,G    - SOS matrix and gain vector G
% Fs       - sample rate
% nfft     - FFT length for analisys
% output:
% coef     - vector with coefficients
%-------------------------------------------------
function [coef,scale]=cvtsos_iir(SOS,G,Fs,nfft)
% convert SOS+G to coefficients of IIR filter
sz=size(SOS);
M=sz(1);
coef=[];
f=(0:nfft-1)/nfft*(Fs/2);
tf0=sos2freqz(SOS,G,nfft);
Gtotal=prod(G);
G=ones(1,M+1);

% define gain for each stage to provide maximim of tf for intermediate output <=0.5
for m=1:M
    tf=sos2freqz(SOS(1:m,:),[G(1:m) 1],nfft);
    tfmax=max(abs(tf));
    G(m)=min(1,0.5/tfmax);
end
for m=1:M
    SOS(m,1:3)= SOS(m,1:3)*G(m);
end
% define last stage shift
dg=Gtotal/prod(G);
scale=round(log2(dg));
% correct coefficient of the last stage
d=pow2(1,scale)/dg;
G(M)=G(M)/d;
% output b,a
coef=SOS'; coef(4,:)=[]; coef=reshape(coef,1,numel(coef));
scale= int32(scale);

% check results and plot final filter response
sos=reshape(double(coef),5,M);
sos=sos';
sos=[sos(:,1:3) ones(M,1) sos(:,4:5)];
g(1:M)=ones(1,M);
g(M+1)=pow2(1,double(scale));
tf=sos2freqz(sos,g,nfft);
plot(f,20*log10(abs(tf)),f,20*log10(abs(tf0))); ylim([-80 0]); title('transfer function, dB'); grid
on;

% convert SOS/G to frequency response
function [tf]=sos2freqz(SOS,G,nfft)
sz=size(SOS);
M=sz(1);
[b,a]=sos2tf(SOS(1,:),[G(1) G(end)]);
tf=freqz(b,a,nfft);
for m=2:M
    [b,a]=sos2tf(SOS(m,:),[G(m) 1]);
    tf=tf.*freqz(b,a,nfft);
end
```

4.2 Matlab Code for Generation the Twiddle Tables

FFT with optimized memory usage require external twiddle tables. Matlab code below shows how to generate twiddles for different functions.

4.2.1 Twiddles for fft_cplx32x16_ie, ifft_cplx32x16_ie, fft_real32x16_ie, ifft_real32x16_ie, fft_cplx16x16_ie, ifft_cplx16x16_ie, fft_real16x16_ie, ifft_real16x16_ie

```
function [twd]=twd32x16_ie(N)
twd = exp(-2j*pi*[1;2;3]*(0:N/4-1)/N);
```

```
twd=twd.';
twd = reshape([imag(twd(:).');real(twd(:).')],1,2*numel(twd));
twd = int16(round(pow2(twd,15)));
```

4.2.2 Twiddles for fft_cplx32x32_ie, ifft_cplx32x32_ie, fft_real32x32_ie, ifft_real32x32_ie

```
function [twd]=twd32x32_ie(N)
twd = exp(-2j*pi*[1;2;3]*(0:N/4-1)/N);
twd = reshape([real(twd(:).');imag(twd(:).')],1,2*numel(twd));
twd = int32(round(pow2(twd,31)));
```

4.2.3 Twiddles for fft_cplxf_ie, ifft_cplxf_ie, fft_realf_ie, ifft_realf_ie

```
function [twd]=twdf_ie(N)
twd = exp(-2j*pi*[1;2;3]*(0:N/4-1)/N);
twd = reshape([real(twd(:).');imag(twd(:).')],1,2*numel(twd));
```

4.3 Matlab Code for Nonlinear Image Normalization

Below is example Matlab code to generate tables for 8-bit and 16-bit nonlinear image normalization using sigmoid transformation.

8-bit variant:

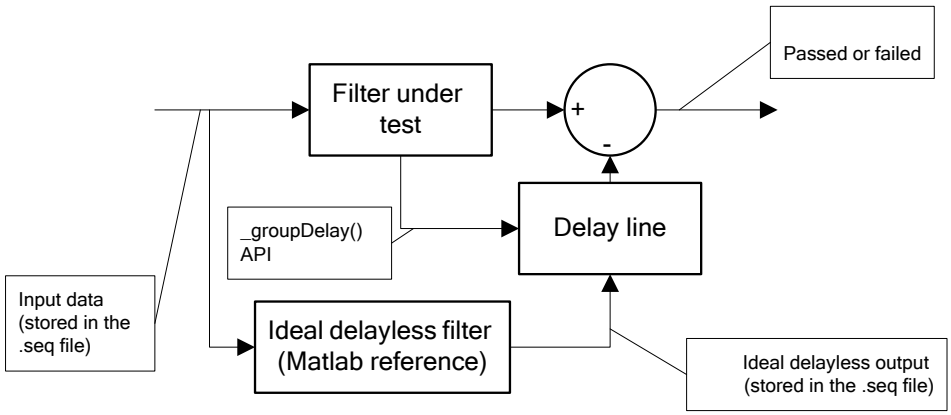
```
x=((0:63)+0.5)/64;
alpha=10;  % density factor
y=1./(1+exp(-(x*alpha-alpha/2)));
tbl=uint8(round(y*256));
```

16-bit variant:

```
x=((0:63)+0.5)/64;
alpha=10;   % density factor
y=1./(1+exp(-(x*alpha-alpha/2)));
tbl=int16(round(y*32768));
```

4.4 About IIR filter validation

Biquad filters may introduce algorithmic delay and amount of this delay is requesting by corresponding API for each specific filter. However, for the testing of all kinds of biquad filters we use **non-delayed** reference data taken from Matlab ideal filters. In fact, the same data are used both for filters with and without algorithmic delay. In the test harness the groupDelay() API is called and manages the extra time shift at the output of ideal output. This shifted reference are comparing with filter under test output and test harness tool decides if test passed or not by checking against threshold. The picture below illustrates this process:


Passed or failed
Filter under test
+
-
_groupDelay() API
Delay line
Input data (stored in the .seq file)
Ideal delayless filter (Matlab reference)
Ideal delayless output (stored in the .seq file)

5 Customer Support

If you have questions, want to report problems or suggestions regarding the **NatureDSP Signal** library or want to port this library to another platforms, contact **IntegrIT** Ltd. at support@integrit.com. Visit www.integrit.com to get more information about products and services.