Gedae E Library Development Kit

The Gedae embeddable function library is constructed as shown in Figure 1. The embeddable primitives supplied with Gedae call a set of core vector functions whenever possible. These core functions are referred to as the E functions. The E functions are a set of algorithms that are typically supported by many vendors in their optimized libraries. If the embeddable graph is targeted to the “host,” then a vanilla C code version of the E function is used. If the graph is targeted to embedded hardware, then the E function references the matching optimized vector routine from the vendor’s library.

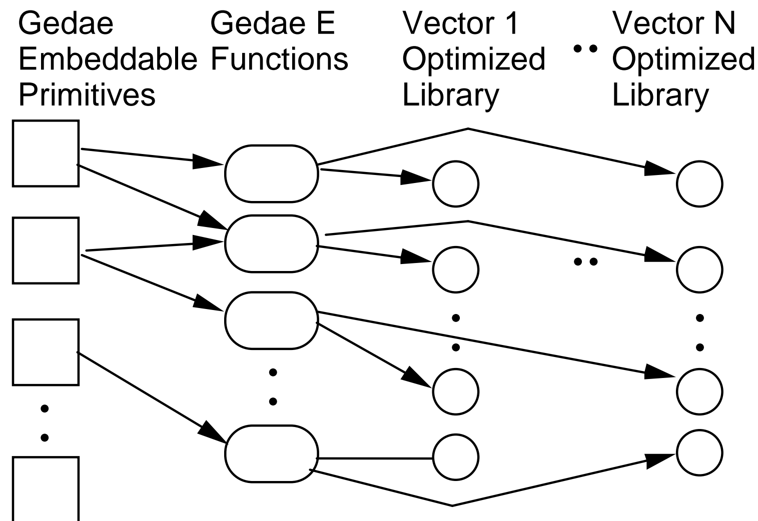

Figure 1 Embeddable Gedae primitive structure

This structure provides a convenient way to integrate with multiple optimized libraries depending on what library is supported by the vendor. A version of the E library is built for each supported target, and during the link process, the appropriate version is linked in. Not all embeddable primitives reference E functions and not all vendor optimized functions are referenced by the E library.

Implementation

These are the set of 300+ functions used by the Gedae primitives. Each of these functions uses generic code that should be able to compile and run on any embedded system. The generic C code versions of these functions are located in:

```
$GEDEV/application/source/embeddable
```

The corresponding header files are located in:

```
$GEDEV/application/include/embeddable
```

For example, here’s the file “e_vadd.c”:

```
#include <e_vadd.h>
#include <dcalc.h>

void e_vadd(float *a,int ia,float *b,int ib,float *c,int ic,int n) {
  int i;
  for (i=0; i<n; i++) {
    *c = *a + *b;
    a += ia;
    b += ib;
    c += ic;
  }
  dcalc_exec("e_vadd",(a,ia,b,ib,c,ic,n));
}
```

This function is able to handle different vector sizes and different strides. Most E functions are built to be versatile in this way, as they are used in a variety of primitives.

For an embedded implementation of the E function, the call to the dcalc_exec() function is not used or needed. This function is used by Gedae-Simulation to approximate the runtime for the E library function call. It is removed from the compilation in Gedae when running outside of the simulation environment on either the host or target processors.

Since you have ported to a new target system, you should take advantage of specialized vector routines included with the embedded system, if they are available. Suppose you wanted to create the optimized version of the e_vadd() function. There are two options: 1) create a header file where you #define the call to the E function to a call to a function from an optimized library, or 2) rewrite the C code so it is optimized for your target processor.

To create only an include file, first copy the header to the embeddable include directory:

Solaris/Linux

```
> cd $GEDEV/application/include/<system_name>
> cp ../embeddable/e_vadd.h .
```

Windows

```
> cd %GEDEV%\application\include\<system_name>
> copy ..\embeddable\e_vadd.h .
```

The header file you just copied over has a function prototype for the E function such as:

```
void e_vadd(float *a,int ia,float *b,int ib,float *c,int ic,int n);
```

Use this prototype to create your #define macro. For example, if your strided add routine is called mathlib_add(), create a header file that contains

```
#define e_vadd(a,ia,b,ib,c,ic,n) mathlib_add(a,ia,b,ib,c,ic,n)
```

To create your own C code for the E function, copy the C code to the embeddable source directory instead of the header file:

Solaris/Linux

```
> cd $GEDEV/application/source/<system_name>
> cp ../embeddable/e_vadd.c .
```

Windows

```
> cd %GEDEV%\application\source\<system_name>
> copy ..\embeddable\e_vadd.c .
```

Then optimize this C code for your target architecture. You only need to implement target-specific versions of the E functions you want to optimize. Each embedded system has it own application source directory. If the function is not provided in the application source directory, the Gedae make system will simply use the vanilla C version provided in the embeddable directory.

Compilation

After you have created E functions for the routines you wish to optimize, rebuild the libraries by rerunning the compilePort script with the ELIB_ONLY flag. The compilePort script is generated for you during the use of the BSP Development Kit and its name is formed based on the name of the processor types for both the emedded host and target.

Solaris/Linux

```
> cd $GEDEV
> perl compilePort_<emb_host>_<emb_target> CLEAN
> perl compilePort_<emb_host>_<emb_target> ELIB_ONLY
```

Windows

```
> cd %GEDEV%
> perl compilePort_<emb_host>_<emb_target> CLEAN
> perl compilePort_<emb_host>_<emb_target> ELIB_ONLY
```

Testing

A suite of test graphs is provided for testing the E functions. These test graphs run the E function on both the host and target processors and compare the results. The graphs are located in the FGlibraries/boxes/testing/ports/e_functions directory and are named after the E function they test, such as the E_vadd graph shown in Figure 2. The test graphs create two branches running the same algorithm then compare the results and exit. To run the test graph individually, simply map one of the two branches to the target processor. Note the two branches of the graph must be mapped to different types of processors (host and target) in order for the test to be effective. This mapping can be done automatically, by using the testEfunctions Perl script.

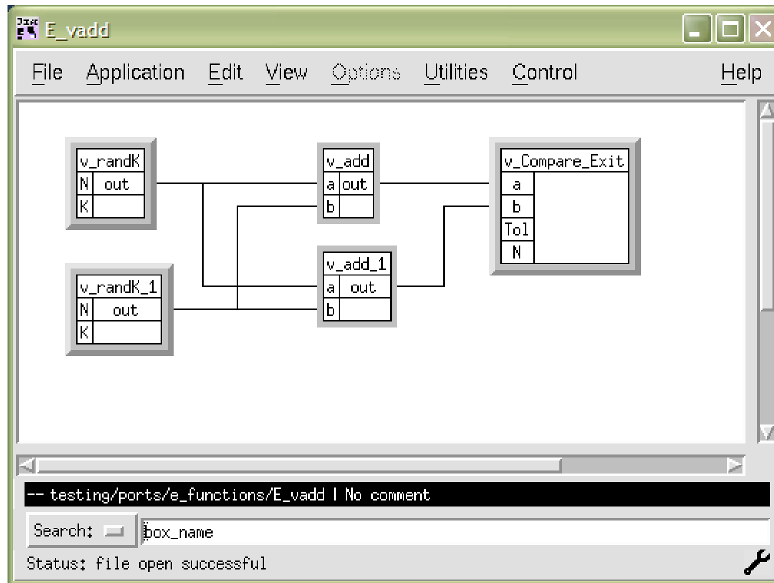

Figure 2 Test graph for the e_vadd function

To run the full test suite, use the testEfunctions Perl script located in the $GEDAE/tools directory. This test script runs each available E function test graph and code generates the group settings necessary to map one of the E functions to the target processor. The test script takes 4 parameters. The first is the operating system, e.g., "redhat", "solaris", or "nt". The second is a text file listing the graphs to run and the boxes in each graph which will be mapped to the target processor. An example file named E_functions_boxes_to_map.txt is provided in the $GEDAE/tools directory. The third parameter is the processor number of the target processor, and the fourth parameter is name of the embedded configuration file that should be used, e.g., to use embedded_config_default, use the parameter "default". For example, to run the test graphs with a standard workstation target configuration, use the following commands:

Solaris/Linux

```
> cd <gedae_user>/<os> # i.e., where the gedae executable is located
> perl $GEDAE\tools\testEfunctions <os>
    $GEDAE\tools\E_functions_boxes_to_map.txt 100 test_e<os>
```

Windows

```
> cd %GEDAEHOME%\nt
> perl %GEDAE%\tools\testEfunctions nt
    %GEDAE%\tools\E_functions_boxes_to_map.txt 100 test_ent
```

In addition, timing graphs are provided for key functions in the library. These functions are dot product, FFT (both split complex and interleaved complex), FIR filter, matrix multiplication, and pointwise multiplication as listed in the table below. The timing graphs can be easily extended to other functions if timing data for those functions is desired.

E Function	Timing Graph
e_rfird	benchmarks/fir/test_fir_timing
e_rmmul	benchmarks/m_mult/test_m_mult_timing
e_dotpr	benchmarks/v_dotpr/test_v_dotpr_timing
e_vmul	benchmarks/v_mult/test_v_mult_timing
e_cvfftb	benchmarks/vx_fft/test_vx_fft_timing
e_zvfftb	benchmarks/vz_fft/test_vz_fft_timing

Each graph has default group settings, so they can be run using the command:

```
> gedae -file benchmarks/<box>/test_<box>_timing -groups default -run
```

These default group settings map the E function to processor 100; this setting can be easily changed in the Map Partition Table of the Group Control. These timing graphs create a text file "e_library_times_<box>.txt" in the user directory with comma-delimited timing data (maximum, minimum, and average) for the functions using several different vector sizes.

Notation

The following charts list all E functions that can be implemented. In these charts, we have marked all functions which have a test graph with an apostrophe (e.g., `e_vadd'`), and we have marked all functions which have a timing graph by using italics (e.g., *`e_vmul`*). Note E functions that are not used in the Gedae embeddable box library do not have a test graph; most developers will choose not to implement optimized versions of functions that do not have a test graph.

Floating Point Functions

Function	Function Prototype	Description
e_avfloat	void e_avfloat(char *a,int ia,float *c,int ic,int n)	Convert from chars to floats c[i*ic] =(float)a[i*ia]
e_dotpr'	void e_dotpr(float *a,int ia,float *b,int ib,float *c,int n)	Dot product c = SUM(a[ia*j]*b[ib*j])
e_maxmgv'	void e_maxmgv(float *a,int ia,float *mv,int *mi,int n)	Maximum magnitude and its first index mv = maximum |a[i*ia]| for i=0..n-1 mi = first value of i such that mv = a[i*ia]
e_maxv'	void e_maxv(float *a,int ia,float *c,int *ic,int n)	Maximum value and its first index c = maximum a[i*ia] for i=0..n-1 ic = first value of i such that c = a[i*ia]
e_maxv0'	void e_maxv0(float *a,int ia,float *c,int n)	Maximum element c = MAX(a[ia*j])
e_meamgv'	void e_meamgv(float *a,int ia,float *c,int n)	Mean of vector elements' magnitudes c = SUM(ABS(a[ia*j])) / n
e_meanv'	void e_meanv(float *a,int ia,float *c,int n)	Mean of vector c = SUM(a[ia*j]) / n
e_measqv'	void e_measqv(float *a,int ia,float *c,int n)	Mean of vector elements squared c = SUM(a[ia*j]^2) / n
e_minmgv'	void e_minmgv(float *a,int ia,float *mv,int *mi,int n)	Minimum magnitude and its first index mv = minimum |a[i*ia]| for i=0..n-1 mi = first value of i such that mv = a[i*ia]
e_minsrt'	void e_minsrt(float *a, int ax, int ay, float *b, int bx, int by, int ix, int iy)	Matrix insert b[ix+i][iy+j] = a[i][j] for i=0..(ax-1), j=0..(ay-1)
e_minv'	void e_minv(float *a,int ia,float	Minimum value and its first index

Function	Function Prototype	Description
	*c,int *ic,int n)	c = minimum a[i*ia] for i=0..n-1 ic = first value of i such that c = a[i*ia]
e_minv0’	void e_minv0(float *a,int ia,float *c,int n)	Minimum element c = MIN(a[ia*j])
e_mtran’	void e_mtran(float *in,float *out,int n,int m)	Matrix transpose (not inplace) b[k][j] = a[j][k] for j=0..(n-1), k=0..(m-1)
e_mtran_ip	void e_mtran_ip(float *in,char *map,int n,int m)	Inplace matrix transpose in[k][j] = in[j][k] for j=0..(n-1), k=0..(m-1) where map is an allocated buffer of size N*M*sizeof(char)
e_mtran2	void e_mtran2(float *in,float *out,int ncin,int nrin,int tcin,int trin)	Matrix transpose of submatrix (not inplace) out_re[i][j]=in_re[i][j] out_im[i][j]=in_im[i][j] for j=0..(nrin-1), i=0..(ncin-1) where submatrix is size nrin x ncin and matrix is size trin x tcin
e_rfird’	void e_rfird(float *a,float *b,int nb,float *c,int d,int n)	FIR filter with decimation c[i] = SUM(a[i*d+j]*b[j]) for i=0..(n-1), j=0..(nb-1)
e_rmmul’	void e_rmmul(float *a,float *b,float *c,int nrc,int ncc,int nca)	Matrix multiplication c[i][k] = SUM(a[i][j]*b[j][k]) for i=0..(nrc-1), j=0..(nca-1), k=0..(ncc-1)
e_rmsqv’	void e_rmsqv(float *a,int ia,float *c,int n)	Vector root mean square c = SQRT(SUM(a[ia*j]^2) / n)
e_sacorr’	void e_sacorr (float *x, int ix, float *r, int ir, int lags, int n)	Auto correlation r[ir*i] = SUM(x[ix*j] * x[ix*(i+j)])

Function	Function Prototype	Description
		for i=0..(lags-1), j=0..(n-i-1)
e_sccoh’	void e_sccoh (float *x, int ix, float *y, int iy, float *r, int ir, int lags, int n)	Cross coherence r[ir*i] = SUM(x[ix*j]*y[iy*(i+j)] – MEAN(x[ix*i])*MEAN(y[iy*i]))
e_sccorr’	void e_sccorr (float *x, int ix, float *y, int iy, float *r, int ir, int lags, int n)	Cross correlation of two vectors r[ir*i] = SUM(x[ix*j] * y[iy*(i+j)]) for i=0..(lags-1), j=0..(n-i-1)
e_scorr’	void e_scorr (float *x, int ix, float *y, int iy, float *r, int n)	Correlation coefficient r = SUM(x[ix*j]*y[iy*j]) / SQRT(SUM(x[ix*j]^2) * SUM(y[iy*j]^2))
e_sfact’	void e_sfact (int *in,int ins,int *out,int outs,int n)	Factorial out[outs*j] = in[ins*j]!
e_slsq’	void e_slsq (float *x, int ix, float *y, int iy, float *slope, float *inter, int n)	Linear least squares fit Find m and b such that NORM(y[iy*j]-(slope*x[ix*j]+ inter)) is minimized
e_smeandev’	void e_smeandev (float *x, int ix, float *r, int n)	Mean deviation r = SUM(ABS(x[ix*j] - MEAN(x[ix*i])) / n
e_srange’	void e_srange (float *x, int ix, float *r, int n)	Distance between largest and smallest vector elements r = MAX(x[ix*j]) - MIN(x[ix*j])
e_sstddev’	void e_sstddev (float *x, int ix, float *r, int n)	Standard deviation r = STDDEV(x[ix*j])
e_subsqs	void e_subsqs(float *a,int ia,float *b, int ib, float *c,int n)	Sum of the squares of the difference of two vectors c = SUM((a[ia*j] - b[ib*j])^2)
e_svar’	void e_svar (float *x, int ix, float *r, int n)	Variance r = (n*SUM(x[ix*j]^2) – SUM(x[ix*j])^2) / (n^2-n)

Function	Function Prototype	Description
e_svdiv’	void e_svdiv(float a,float *b,int ib,float *c,int ic,int n)	Division of a scalar by a vector c[ic*j] = a / b[ib*j]
e_sve’	void e_sve(float *a,int ia,float *c,int n)	Sum of vector elements c = SUM(a[ia*j])
e_svemg’	void e_svemg(float *a,int ia,float *c,int n)	Sum of vector elements' magnitudes c = SUM(ABS(a[ia*j]))
e_svesq’	void e_svesq(float *a,int ia,float *c,int n)	Sum of vector elements squared c = SUM(a[ia*j]^2)
e_svessq’	void e_svessq(float *a,int ia,float *c,int n)	Sum of vector elements sign-squared c = SUM(ABS(a[ia*j])*a[ia*j])
e_svfloat	void e_svfloat(short *a,int ia,float *c,int ic,int n)	Convert from shorts to floats c[i*ic] =(float)a[i*ia]
e_svsub’	void e_svsub(float a,float *b,int ib,float *c,int ic,int n)	Subtraction of a vector from a scalar c[ic*j] = a - b[ib*j]
e_swmean’	void e_swmean(float *x, int ix, float *w, int iw, float *r, int n)	Weighted mean r = SUM(x[ix*j]*w[iw*j]) / SUM(w[iw*j])
e_uavfloat	void e_uavfloat(unsigned char *a,int ia,float *c,int ic,int n)	Convert from unsigned chars to floats c[i*ic] =(float)a[i*ia]
e_usvfloat	void e_usvfloat(unsigned short *a,int ia,float *c,int ic,int n)	Convert from unsigned shorts to floats c[i*ic] =(float)a[i*ia]
e_uivfloat	void e_uivfloat(unsigned int *a,int ia,float *c,int ic,int n)	Convert from unsigned ints to floats c[i*ic] =(float)a[i*ia]
e_vaa’	void e_vaa(float *a,int ia,float *b,int ib,float *c,int ic,float *d, int id,int n)	Addition of three vectors d[id*j] = a[ia*j] + b[ib*j] + c[ic*j]
e_vaam’	void e_vaam(float *a,int ia,float *b, int ib,float *c,int ic, float *d, int id,float *e, int ie,int n)	Addition of two pairs of vectors and multiply the results e[ie*j] = (a[ia*j] + b[ib*j]) *(c[ic*j]+d[id*j])

Function	Function Prototype	Description
e_vabs’	void e_vabs(float *a,int ia,float *c,int ic,int n)	Absolute value of vector elements c[ic*j] = ABS(a[ia*j])
e_vacos’	void e_vacos(float *a,int ia,float *c,int ic,int n)	Arccosine of vector elements c[ic*j] = ARCCOS(a[ia*j])
e_vadd’	void e_vadd(float *a,int ia,float *b,int ib,float *c,int ic,int n)	Addition of two vectors c[ic*j] = a[ia*j] + b[ib*j]
e_vafix	void e_vafix(float *a,int ia,char *c,int ic,int n)	Convert from floats to chars c[i*ic] =(char)a[i*ia]
e_vam’	void e_vam(float* a,int ia,float *b,int ib,float *c,int ic,float *d, int id,int n)	Vector addition and multiplication d[id*j] = (a[ia*j]+b[ib*j]) * c[ic*j]
e_vasbm’	void e_vasbm(float *a,int ia,float *b, int ib,float *c,int ic, float *d,int id,float *e, int ie, int n)	Addition of one pair of vectors, subtraction of another pair, and multiply the results e[ie*j] = (a[ia*j]+b[ib*j]) * (c[ic*j]-d[id*j])
e_vasin’	void e_vasin(float *a,int ia,float *c,int ic,int n)	Arcsine of vector elements c[ic*j] = ARCSIN(a[ia*j])
e_vasm’	void e_vasm(float *a,int ia,float *b,int ib,float c,float *d,int id, int n)	Vector addition and scalar multiplication d[id*j] = (a[ia*j]+b[ib*j]) * c
e_vasub’	void e_cvasub(float *a,int ia,float *b,int ib,float *c,int ic,float *d,int id,int n)	Add two vectors and subtract away another vector d[id*j] = a[ia*j] + b[ib*j] - c[ic*j]
e_vatan’	void e_vatan(float *a,int ia,float *c,int ic,int n)	Arctangent of vector elements c[ic*j] = ARCTAN(a[ia*j])
e_vatan2’	void e_vatan2(float *a,int ia,float *b, int ib,float *c,int ic, int n)	Divide vector elements and take arctangent c[ic*j] = ARCTAN(a[ia*j]/ b[ib*j])
e_vceil	void e_vceil(float *a,int ia,float	Ceiling

Function	Function Prototype	Description
	*c,int ic,int n)	c[ic*j] = CEILING(a[ia*j])
e_vclip'	void e_vclip(float *a, int ia, float b, float c, float *d,int id, int n)	Clip elements of an vector so values are between min and max d[id*j] = MAX(b,MIN(c,a[ia*j]))
e_vclr'	void e_vclr(float *c, int ic, int n)	Set vector to zeros c[ic*j] = 0
e_vcos'	void e_vcos(float *a,int ia,float *c,int ic,int n)	Cosine of vector elements c[ic*j] = COS(a[ia*j])
e_vcosh'	void e_vcosh (float *a,int ia,float *b,int ib,int n)	Hyperbolic cosine of vector elements b[ib*j] = COSH(a[ia*j])
e_vcub'	void e_vcub(float *a,int ia,float *b,int ib,int n)	Cube vector elements b[ib*j] = a[ia*j]^3
e_vdct	void e_vdct(float *in,float *out,int n,int d)	Compute real Discrete Cosine Transform if (d) out = DCT(in) else out = IDCT(in)
e_dctwts	int e_set_dctwts(int n) int e_dctwts(void)	Allocate real FFT buffers Use in conjunction with e_vdct
e_vdiv'	void e_vdiv(float *a,int ia,float *b,int ib,float *c,int ic,int n)	Division of two vectors c[ic*j] = a[ia*j] / b[ib*j]
e_veql'	void e_veql(float *a,int ia,float *b,int ib,int *c,int n)	Evaluates if two complex vectors are equal c = AND(a[ia*j] == b[ib*j])
e_vexp'	void e_vexp(float *a,int ia,float *c, int ic,int n)	Vector exponential c[ic*j] = e^(a[ia*j])
e_vexp10'	void e_vexp10(float *a,int ia,float *c,int ic,int n)	Vector base 10 exponential c[ic*j] = 10^(a[ia*j])
e_vexp2'	void e_vexp2(float *a,int ia,float *c,int ic,int n)	Vector base 2 exponential c[ic*j] = 2^(a[ia*j])
e_vfill'	void e_vfill(float a, float *c,int ic,int n)	Vector fill c[ic*j] = a
e_vfirst'	void e_vfirst(float *a,int ia,int	Index of first non-zero vector element

Function	Function Prototype	Description
	*c,int n)	a[c] != 0 AND a[j] == 0 for j=0..(c-1)
e_vfloor	void e_vfloor(float *a,int ia,float *c,int ic,int n)	Floor c[ic*j] = floor(a[ia*j])
e_vgathr	void e_vgathr(float *a,int *b,int ib,float *c,int ic,int n)	Gather c[ic*j] = a[b[ib*j]]
e_vhypot'	void e_vhypot(float *a,int ia,float *b,int ib,float *c,int ic,int n)	Elementwise square root of sum of squares of two vectors c[ic*j] = SQRT(a[ia*j]+b[ib*j])
e_vifix'	void e_vifix(float *a,int ia,int *c,int ic,int n,int f)	Conversion to integer by truncation and rounding if (f) c[ic*j] = (int)(a[ia*j]) else c[ic*j] = ROUND(a[ia*j])
e_vklip'	void e_vklip(float *a,int ia,float b, float *c,int ic,int n)	Clip elements of vector so that values are between +/- ABS(b) c[ic*j] = MAX(-ABS(b), MIN(ABS(b),a[ia*j]))
e_vlast'	void e_vlast(float *a,int ia,int *c,int n)	Index of last non-zero vector element a[c] != 0 AND a[j] == 0 for j=(c+1)..(n-1)
e_vlim'	void e_vlim(float *a,int ia,float b, float c,float *d,int id,int n)	Vector limit to single absolute value, the sign of which depends on whether the input exceeds a threshold if (a[ia*j] < b) d[id*j] = -c else d[id*j] = c
e_vlint'	void e_vlint(float *t,int nt,float *u, int ius,float *y,int iys,int n)	Vector linear interpolate y[iys*j] = t[nt] + (u[ius*j]-nt)*(t[nt+1]-t[nt]) where nt=(int)u[ius*j]
e_vlog'	void e_vlog(float *a,int ia,float *c, int ic,int n)	Natural logarithm of vector elements c[ic*j] = LN(a[ia*j])

Function	Function Prototype	Description
e_vlog10'	void e_vlog10(float *a,int ia,float *c, int ic,int n)	Logarithm base 10 of vector elements c[ic*j] = LOG10(a[ia*j])
e_vlog2'	void e_vlog2(float *a,int ia,float *c, int ic,int n)	Logarithm base 2 of vector elements c[ic*j] = LOG2(a[ia*j])
e_vma'	void e_vma(float* a,int ia,float *b,int ib, float *c,int ic, float *d, int id,int n)	Vector multiplication and addition d[id*j] = (a[ia*j]*b[ib*j]) + c[ic*j]
e_vmax'	void e_vmax(float *a,int ia,float *b, int ib,float *c,int ic, int n)	Elementwise maximum of two vectors c[ic*j] = MAX(a[ia*j],b[ib*j])
e_vmaxmg'	void e_vmaxmg(float *a,int ia,float *b, int ib,float *c,int ic, int n)	Elementwise maximum magnitude of two vectors c[ic*j] = MAX(ABS(a[ia*j]),ABS(b[ib*j]))
e_vmin'	void e_vmin(float *a,int ia,float *b, int ib,float *c,int ic,int n)	Elementwise minimum of two vectors c[ic*j] = MIN(a[ia*j],b[ib*j])
e_vminmg'	void e_vminmg(float *a,int ia,float *b, int ib,float *c,int ic, int n)	Elementwise minimum magnitude of two vectors c[ic*j] = MIN(ABS(a[ia*j]),ABS(b[ib*j]))
e_vmma'	void e_vmma(float *a,int ia,float *b, int ib,float *c,int ic, float *d,int id,float *e, int ie,int n)	Multiplication of two pairs of vectors and add results e[ie*j] = (a[ia*j]*b[ib*j]) + (c[ic*j]*d[id*j])
e_vmmsb'	void e_vmmsb(float *a,int ia,float *b, int ib,float *c,int ic,float *d,int id,float *e,int ie, int n)	Multiplication of two pairs of vectors and subtract results e[ie*j] = (a[ia*j]*b[ib*j]) - (c[ic*j]*d[id*j])
e_vmov'	void e_vmov(float *a,int ia,float *c, int ic,int n)	Move vector c[ic*j] = a[ia*j]

Function	Function Prototype	Description
e_vmrg’	void e_vmrg(float *a,int ia,float *b, int ib,float *c,int ic,int m, int n)	Merge two sorted vectors into one sorted vector if (a[ia*i] < b[ib*j]) c[ic*k++] = a[ia*i++] else c[ic*k] = b[ib*j++]
e_vmsa’	void e_vmsa(float *a,int ia,float *b, int ib,float c,float *d, int id,int n)	Vector multiplication and scalar addition d[id*j] = a[ia*j]*b[ib*j] + c
e_vmsb’	void e_vmsb(float *a,int ia,float *b,int ib,float *c,int ic,float *d,int id,int n)	Vector multiplication and subtraction d[id*j] = (a[ia*j]*b[ib*j]) - c[ic*j]
e_vmsn’	void e_vmsn(float *a,int ia,float *b,int ib,float *c,int ic,float *d,int id,int n)	Subtract product of two vectors from a third vector d[id*j] = c[ic*j] - (a[ia*j]*b[ib*j])
e_vmul’	void e_vmul(float *a,int ia,float *b,int ib,float *c,int ic,int n)	Multiplication of two vectors c[ic*j] = a[ia*j] * b[ib*j]
e_vnabs’	void e_vnabs(float *a,int ia,float *c,int ic,int n)	Vector negative absolute value c[ic*j] = -ABS(a[ia*j])
e_vneg’	void e_vneg(float *a,int ia,float *c,int ic,int n)	Vector negate c[ic*j] = -a[ia*j]
e_vneql’	void e_vneql(float *a,int ia,float *b,int ib,int *c,int n)	Evaluates if two vectors are not equal c = OR(a[ia*j] != b[ib*j])
e_vpolre	void e_vpolrect (float *r,int ir,float *theta, int itheta, float *x,int ix,float *y,int iy,int n)	Convert from polar to rectangular coordinates c[j] = a[j]*cos(b[j]), d[j] = a[j]*sin(b[j])
e_vpow’	void e_vpow(float *a,int ia,float *b,int ib,float *c,int ic,int n)	Vector power c[ic*j] = a[ia*j]^b[ib*j]
e_vprog’	void e_vprog(float *a,int ia,float *c,int ic,int n)	Vector progression c[ic*j] = SUM(a[ia*i]) for i=0..j
e_vramp’	void e_vramp(float a,float b,float	Vector ramp

Function	Function Prototype	Description
	*c,int ic,int n)	c[ic*j] = a + j*b
e_vrecip'	void e_vrecip(float *a,int ia,float *c,int ic,int n)	Take the reciprocal of a vector c[ic*j] = 1 / a[ia*j]
e_vrectp	void e_vrectpol (float *x,int ix, float *y, int iy, float *r, int ir, float *theta, int itheta, int n)	Convert from rectangular to polar coordinates c[j] = SQRT(a[j]^2+b[j]^2), d[j] = ATAN(b[j] / a[j])
e_vrfft'	void e_vrfft(float *in,float *out,int n,int d)	Compute real Fast Fourier Transform if (d) out = FFT(in) else out = IFFT(in) Packed into n/2 complex values IM(out[0]) = Nyquist Point
e_vrfftnd'	e_vrfftnd(float *in,float *out,int n,int d)	FFT similar to e_vrfft that may or may not scale results
e_rfftwts	int e_set_rfftwts(int n) int e_rfftwts(void)	Allocate real FFT buffers Use in conjunction with e_vrfft
e_vround	void e_vround(float *a,int ia,float *c,int ic,int n)	Round if (a[ia*j] < 0) c[ic*j] = (float)(int)(a[ia*j]-0.5) else c[ic*j] = (float)(int)(a[ia*j]+0.5)
e_vrvrs'	void e_vrvrs(float *c, int ic, int n)	Vector reverse inplace SWAP(c[ic*j], c[ic*(n-1-j)]) for j=0..floor(n/2)
e_vsadd'	void e_vsadd(float *a,int ia,float b,float *c,int ic,int n)	Addition of vector and scalar c[ic*j] = a[ia*j] + b
e_vsam'	void e_vsam(float *a,int ia,float b, float *c,int ic,float *d,int id,int n)	Vector scalar addition and multiplication d[id*j] = (a[ia*j]+b) * c[ic*j]
e_vsbm'	void e_vsbm(float *a,int ia,float	Vector subtraction and multiplication

Function	Function Prototype	Description
	*b, int ib,float *c,int ic,float *d, int id, int n)	d[id*j] = (a[ia*j]-b[ib*j]) * c[ic*j]
e_vsbsbm'	void e_vsbsbm(float *a,int ia,float *b, int ib,float *c,int ic, float *d,int id,float *e, int ie,int n)	Subtraction of two pairs of vectors and multiply the results e[ie*j] = (a[ia*j]-b[ib*j]) * (c[ic*j]-d[id*j])
e_vsbsm'	void e_vsbsm(float *a,int ia,float *b, int ib,float c,float *d, int id, int n)	Vector subtraction and scalar multiplication d[id*j] = (a[ia*j]-b[ib*j]) * c
e_vscale'	void e_vscale (float *a,int ia, long k, float *b,int ib,int n)	Scale a vector by a power of 2 b[ib*j] = a[ia*j] * 2^k
e_vscatr	void e_vscatr(float *a, int ia, int *b,int ib,float *c,int n)	Scatter c[b[ib*j]] = a[ia*j]
e_vsdiv'	void e_vsdiv(float *a,int ia,float b,float *c,int ic,int n)	Division of vector by scalar c[ic*j] = a[ia*j] / b
e_vsfix	void e_vsfix(float *a,int ia,short *c,int ic,int n)	Convert from floats to shorts c[i*ic] =(short)a[i*ia]
e_vshrink'	void e_vshrink(float *a,int ia,float *b, int ib, int n)	Average vector elements pairwise, cutting vector length in half b[ib*j] = 0.5 * (a[ia*(2*i)] + a[ia*(2*i+1)])
e_vsin'	void e_vsin(float *a,int ia,float *c, int ic,int n)	Sine of vector elements c[ic*j] = SIN(a[ia*j])
e_vsinh'	void e_vsinh (float *a,int ia,float *b, int ib,int n)	Hyperbolic sine of vector elements b[ib*j] = SINH(a[ia*j])
e_vsma'	void e_vsma(float *a,int ia,float b, float *c,int ic,float *d,int id,int n)	Vector scalar multiplication and addition d[id*j] = a[ia*j]*b + c[ic*j]
e_vsmsa'	void e_vsmsa(float *a,int ia,float b, float c,float *d,int id, int n)	Vector scalar multiplication and scalar addition d[id*j] = a[ia*j]*b + c

Function	Function Prototype	Description
e_vsmsb’	void e_vsmsb(float *a,int ia,float b, float *c,int ic,float *d, int id,int n)	Vector scalar multiplication and subtraction d[id*j] = a[ia*j]*b - c[ic*j]
e_vsmul’	void e_vsmul(float *a,int ia,float b, float *c,int ic,int n)	Multiplication of vector by scalar c[ic*j] = a[ia*j] * b
e_vsort’	void e_vsort(float *c, int n, int f)	Sort vector inplace in ascending (f==1) or descending order (f==-1)
e_vsq’	void e_vsq(float *a,int ia,float *c,int ic,int n)	Square vector elements c[ic*j] = a[ia*j]^2
e_vsqrt’	void e_vsqrt(float *a,int ia,float *c,int ic,int n)	Square root of vector elements c[ic*j] = a[ia*j]^2
e_vss’	void e_vss(float *a,int ia,float *b,int ib,float *c,int ic,float *d, int id, int n)	Subtract two complex vectors from one vector d[id*j] = a[ia*j] - b[ib*j] - c[ic*j]
e_vssq’	void e_vssq(float *a,int ia,float *c, int ic,int n)	Signed square vector elements c[ic*j] = ABS(a[ia*j])*a[ia*j]
e_vsub’	void e_vsub(float *a,int ia,float *b, int ib,float *c,int ic,int n)	Subtraction of two vectors c[ic*j] = a[ia*j] - b[ib*j]
e_vsubsq	void e_vsubsq(float *a,int ia,float *b,int ib,float *c,int ic,int n)	Vector subtract and square result c[ic*j] = (a[ia*j] - b[ib*j])^2
e_vswap’	void e_vswap(float *a,int ia,float *b, int ib,int n)	Swap two vectors SWAP(a[ia*j], b[ib*j])
e_vtan’	void e_vtan(float *a,int ia,float *c, int ic,int n)	Tangent of vector elements c[ic*j] = TAN(a[ia*j])
e_vtanh’	void e_vtanh (float *a,int ia,float *b, int ib,int n)	Hyperbolic tangent of vector elements b[ib*j] = TANH(a[ia*j])
e_vthr’	void e_vthr(float *a, int ia, float b float *c, int ic, int n)	Vector clip to greater than scalar if (a[ia*j] < b) c[ic*j] = b else c[ic*j] = a[ia*j]
e_vthres’	void e_vthres(float *a,int ia,float b, float *c,int ic,int n)	Set vector elements below threshold to zero

Function	Function Prototype	Description
		if (a[ia*j] < b) c[ic*j] = 0 else c[ic*j] = a[ia*j]
e_vtrunc	void e_vtrunc(float *a,int ia,float *c,int ic,int n)	Truncate c[ic*j] = (float)(int)a[ia*j]
e_vuafix	void e_vafix(float *a,int ia,unsigned char *c,int ic,int n)	Convert from floats to unsigned chars c[i*ic] =(unsigned char)a[i*ia]
e_vuifix	void e_vuifix(float *a,int ia,unsigned int *c,int ic,int n)	Convert from floats to unsigned ints c[i*ic] =(unsigned int)a[i*ia]
e_vusfix	void e_vusfix(float *a,int ia,unsigned short *c,int ic,int n)	Convert from floats to unsigned shorts c[i*ic] =(unsigned short)a[i*ia]

Complex Functions

Function	Function Prototype	Description
e_cdotc'	void e_cdotc(complex *a,int ia,complex *b,int ib,complex *c,int n)	Complex dot product of conjugate of first vector with second c =SUM(CONJ(a[j*ia/2])*b[j*ib/2])
e_cdotpr'	void e_cdotpr(complex *a,int ia,complex *b,int ib,complex *c,int n)	Complex dot product c = SUM(a[j*ia/2]*b[j*ib/2])
e_cfird'	void e_cfird(complex *a,complex *b,int nb,complex *c,int d,int n)	FIR filter with decimation c[i] = SUM(a[i*d+j]*b[j]) for i=0..(n-1), j=0..(nb-1)
e_cmmul'	void e_cmmul(complex *a,complex *b,complex *c,int nrc,int ncc,int nca)	Complex matrix multiplication c[i][k] = SUM(a[i][j]*b[j][k]) for i=0..(nrc-1), j=0..(nca-1), k=0..(ncc-1)
e_cmtran'	void e_cmtran(complex *aa,complex *cc,int n,int m,int t)	Complex matrix transpose (not inplace) If (t) c[k][j] = CONJ(a[j][k]) else c[k][j] = a[j][k] for j=0..(n-1), k=0..(m-1)
e_cmtran_ip	void e_mtran_ip(complex *in,char *map,int n,int m)	Inplace matrix transpose in[k][j] = in[j][k] for j=0..(n-1), k=0..(m-1) where map is an allocated buffer of size N*M*sizeof(char)
e_cmtran2'	void e_cmtran2(complex *in,complex *out,int ncin,int nrin,int tcin,int trin)	Complex matrix transpose of submatrix (not inplace) out[i][j]=in[i][j] for j=0..(nrin-1), i=0..(ncin-1) where submatrix is size nrin x ncin and matrix is size trin x tcin

Function	Function Prototype	Description
e_crvadd’	void e_crvadd(complex *a,int ia,float *b,int ib,complex *c,int ic,int n)	Addition of comlex vector and real vector c[j*ic/2] = a[j*ia/2] + b[j*ib]
e_crvdiv’	void e_crvdiv(complex *a,int ia,float *b,int ib,complex *c,int ic,int n)	Division of complex vector by real vector c[j*ic/2] = a[j*ia/2] / b[j*ib]
e_crvmul’	void e_crvmul(complex *a,int ia,float *b,int ib,complex *c,int ic,int n)	Multiplication of complex vector and real vector c[j*ic/2] = a[j*ia/2] * b[j*ib]
e_crvsub’	void e_crvsub(complex *a,int ia,float *b,int ib,complex *c,int ic,int n)	Subtract real vector from complex vector c[i*ic/2] = a[i*ia/2] - b[i*ib]
e_csvemg’	void e_csvemg(complex *a,int ia,float *c,int n)	Complex sum of vector element magnitudes c = SUM(ABS(a[j*ia/2]))
e_csvemgs’	void e_csvemgs(complex *a,int ia,float *c,int n)	Complex sum of vector element magnitudes squared c = SUM(ABS(a[j*ia/2])^2)
e_csvmul’	void e_csvmul(complex *a,complex *b,int ib,complex *c,int ic,int n)	Multiplication of complex vector by complex scalar c[j*ic/2] = a * b[j*ib/2]
e_cvabs’	void e_cvabs(complex *a,int ia,float *c,int ic,int n)	Absolute value of complex vector c[ic*j] = ABS(a[j*ia/2])
e_cvadd’	void e_cvadd(complex *a,int ia,complex *b,int ib,complex *c,int ic,int n)	Addition of two complex vectors c[j*ic/2] = a[j*ia/2] + b[j*ib/2]
e_cvasub’	void e_cvasub(complex *a,int ia,complex *b, int ib,complex *c,int ic, complex *d,int id,int n)	Add two complex vectors and subtract away another vector d[j*id/2] = a[j*ia/2] + b[j*ib/2] - c[j*ic/2]
e_cvclip’	void e_cvclip (complex *a, int	Clip elements of a complex vector so

Function	Function Prototype	Description
	ia,complex min,complex max,complex *out, int iout, int n)	values are between min and max RE(out[iout*j]) = MAX(min,MIN(max,RE(a[j*ia/2]))), IM(out[j*iout/2]) = MAX(min,MIN(max,IM(a[j*ia/2])))
e_cvclr'	void e_cvclr(complex *c,int ic,int n)	Set complex vector to zeros c[j*ic/2] = 0+0i
e_cvcomb'	void e_cvcomb(float *a,int ia,float *b,int ib, complex *c,int ic,int n)	Form complex vector from two real vectors c[j*ic/2] = a[j*ia] + i*b[j*ib]
e_cvconj'	void e_cvconj(complex *a,int ia,complex *c,int ic,int n)	Conjugate complex vector c[j*ic/2] = CONJ(a[j*ia/2])
e_cvcub'	void e_cvcub(complex *a,int ia,complex *b,int ib,int n)	Cube vector elements b[j*ib/2] = a[j*ia/2]^3
e_cvdiv'	void e_cvdiv(complex *a,int ia,complex *b,int ib,complex *c,int ic,int n)	Divide one complex vector by another c[j*ic/2] = a[j*ia/2] / b[j*ib/2]
e_cveql'	void e_cveql(complex *a,int ia,complex *b,int ib,int *c,int n)	Evaluates if two complex vectors are equal c = AND(a[j*ia/2] == b[j*ib/2])
e_cvexp'	void e_cvexp(float *a,int ia,complex *c,int ic,int n)	Complex vector exponential c[j*ic/2] = e^(i * a[j*ia])
e_cvfftb'	void e_cvfftb(complex *in,complex *out,int n,int d)	Compute Fast Fourier Transform if (d) out = FFT(in) else out = IFFT(in)
e_fftwts	int e_set_fftwts(int n) int e_fftwts(void)	Allocate FFT buffers Use in conjunction with e_cvfftb
e_cvfill'	void e_cvfill(complex a, complex *c,int ic,int n)	Complex vector fill c[j*ic/2] = a
e_cvgathr	void e_cvgathr(complex *a,int *b,int ib,complex *c,int ic,int n)	Gather c[ic*j] = a[b[ib*j]]

Function	Function Prototype	Description
e_cvifftbnd'	e_cvifftbnd(complex *in, complex *out, int n)	Inverse FFT that may or may not scale results
e_cvmags'	void e_cvmags(complex *a,int ia,float *c, int ic,int n)	Complex vector magnitude squared c[j*ic] = ABS(a[j*ia/2])^2
e_cvmov'	void e_cvmov(complex *a,int ia,complex *c, int ic,int n)	Move complex vector c[j*ic/2] = a[j*ia/2]
e_cvmsn'	void e_cvmsn(complex *a,int ia,complex *b, int ib,complex *c,int ic, complex *d,int id,int n)	Multiply two complex vectors and subtract away another vector d[j*id/2] = c[j*ic/2] - a[j*ia/2] * b[j*ib/2]
e_cvmul'	void e_cvmul(complex *a,int ia,complex *b, int ib,complex *c,int ic,int n, int f)	Multiplication of two complex vectors if (f==-1) c[j*ic/2] = CONJ(a[j*ia/2]) * b[j*ib/2] else c[j*ic/2]=a[j*ia/2]*b[j*ib/2]
e_cvneg'	void e_cvneg(complex *a,int ia,complex *c, int ic,int n)	Negate complex vector c[j*ic/2] = -a[j*ia/2]
e_cvneql'	void e_cvneql(complex *a,int ia,complex *b, int ib,int *c,int n)	Evaluates if two complex vectors are not equal c = OR(a[j*ia/2] != b[j*ib/2])
e_cvphas'	void e_cvphas(complex *a,int ia,float *c, int ic,int n)	Find the phase of the complex vector c[j*ic] = atan(IM(a[j*ia/2]) / RE(b[j*ib/2]))
e_cvramp'	void e_cvramp(complex a, complex b,complex *c, int ic, int n)	Complex vector ramp c[j*ic/2] = a + j*b
e_cvrcip'	void e_cvrcip(complex *a,int ia,complex *c, int ic,int n)	Take the reciprocal of a complex vector c[j*ic/2] = 1 / a[j*ia/2]
e_cvrsmul'	void e_cvrsmul(complex *a,int ia,float b,complex *c,int ic,int n)	Multiplication of complex vector by real scalar c[j*ic/2] = b * a[j*ia/2]
e_cvscatr	void e_cvscatr(complex *a, int ia, int *b,int ib,complex *c,int n)	Scatter c[b[ib*j]] = a[ia*j]

Function	Function Prototype	Description
e_cvsma’	void e_cvsma(complex *a, int ia, complex *b, complex *c,int ic,complex *y, int iy, int n)	Complex vector scalar multiplication and addition y[j*iy/2] = a[j*ia/2]*b+c[j*ic/2]
e_cvsplit’	void e_cvsplit(complex *a,int ia,float *b, int ib,float *c,int ic,int n)	Split complex vector into real and imaginary parts b[j*ib] = RE(a[j*ia/2]), c[j*ic] = IM(a[j*ia/2])
e_cvss’	void e_cvss(complex *a,int ia,complex *b, int ib,complex *c,int ic, complex *d,int id,int n)	Subtract two complex vectors from one vector d[j*id/2] = a[j*ia/2] - b[j*ib/2] - c[j*ic/2]
e_cvsub’	void e_cvsub(complex *a,int ia,complex *b, int ib,complex *c,int ic,int n)	Subtraction of two complex vector c[j*ic/2] = a[j*ia/2] - b[j*ib/2]
e_polar’	void e_polar(complex *a,int ia,complex *c,int ic,int n)	Convert complex vector from rectangular to polar coordinates RE(c[ic*j]) = ABS(a[ia*j]), IM(c[ic*j]) = ATAN(IM(a[ia*j]) / RE(a[ia*j]))
e_rect’	void e_rect(complex *a,int ia,complex *c,int ic,int n)	Convert complex vector from polar to rectangular coordinates c[ic*j] = RE(a[ia*j]) * e^(i*IM(a[ia*j]))

Double Precision Functions

Function	Function Prototype	Description
e_ddotpr’	void e_ddotpr(double *a,int ia,double *b,int ib,double *c,int n)	Dot product c = SUM(a[ia*j]*b[ib*j])
e_dmtran_ip	void e_dmtran_ip(double *in,char *map,int n,int m)	Inplace matrix transpose in[k][j] = in[j][k] for j=0..(n-1), k=0..(m-1) where map is an allocated buffer of size N*M*sizeof(char)
e_dmmul’	void e_dmmul(double *a,double *b,double *c,int nrc,int ncc,int nca)	Matrix multiplication c[i][k] = SUM(a[i][j]*b[j][k]) for i=0..(nrc-1), j=0..(nca-1), k=0..(ncc-1)
e_dsvesq’	void e_dsvesq(double *a,int ia,double *c,int n)	Sum of vector elements squared c = SUM(a[ia*j]^2)
e_dvaa’	void e_dvaa(double *a,int ia,double *b,int ib,double *c,int ic,double *d, int id,int n)	Addition of three vectors d[id*j] = a[ia*j] + b[ib*j] + c[ic*j]
e_dvaam’	void e_dvaam(double *a,int ia,double *b, int ib,double *c,int ic, double *d, int id,double *e, int ie,int n)	Addition of two pairs of vectors and multiply the results e[ie*j] = (a[ia*j] + b[ib*j]) *(c[ic*j]+d[id*j])
e_dvabs’	void e_dvabs(double *a,int ia,double *c,int ic,int n)	Absolute value of vector elements c[ic*j] = ABS(a[ia*j])
e_dvacos’	void e_dvacos(double *a,int ia,double *c,int ic,int n)	Arccosine of vector elements c[ic*j] = ARCCOS(a[ia*j])
e_dvadd’	void e_dvadd(double *a,int ia,double *b,int ib,double *c,int ic,int n)	Addition of two vectors c[ic*j] = a[ia*j] + b[ib*j]
e_dvafix	void e_dvafix(double *a,int ia,char	Convert from doubles to chars

Function	Function Prototype	Description
	*c,int ic,int n)	c[i*ic] =(char)a[i*ia]
e_dvam'	void e_dvam(double* a,int ia,double *b,int ib,double *c,int ic,double *d, int id,int n)	Vector addition and multiplication d[id*j] = (a[ia*j]+b[ib*j]) * c[ic*j]
e_dvasbm'	void e_dvasbm(double *a,int ia,double *b, int ib,double *c,int ic, double *d,int id,double *e, int ie, int n)	Addition of one pair of vectors, subtraction of another pair, and multiply the results e[ie*j] = (a[ia*j]+b[ib*j]) * (c[ic*j]-d[id*j])
e_dvasin'	void e_dvasin(double *a,int ia,double *c,int ic,int n)	Arcsine of vector elements c[ic*j] = ARCSIN(a[ia*j])
e_dvasm'	void e_dvasm(double *a,int ia,double *b,int ib,double c,double *d,int id, int n)	Vector addition and scalar multiplication d[id*j] = (a[ia*j]+b[ib*j]) * c
e_dvasub'	void e_dvasub(complex *a,int ia,complex *b,int ib,complex *c,int ic,complex *d,int id,int n)	Add two vectors and subtract away another vector d[id*j] = a[ia*j] + b[ib*j] - c[ic*j]
e_dvatan'	void e_dvatan(double *a,int ia,double *c,int ic,int n)	Arctangent of vector elements c[ic*j] = ARCTAN(a[ia*j])
e_dvatan2'	void e_dvatan2(double *a,int ia,double *b, int ib,double *c,int ic, int n)	Divide vector elements and take arctangent c[ic*j] = ARCTAN(a[ia*j]/ b[ib*j])
e_dvclip'	void e_dvclip(double *a, int ia, double b, double c, double *d,int id, int n)	Clip elements of an vector so values are between min and max d[id*j] = MAX(b,MIN(c,a[ia*j]))
e_dvclr'	void e_dvclr(double *c, int ic, int n)	Set vector to zeros c[ic*j] = 0
e_dvcos'	void e_dvcos(double *a,int ia,double *c,int ic,int n)	Cosine of vector elements c[ic*j] = COS(a[ia*j])
e_dvcosh'	void e_dvcosh (double *a,int	Hyperbolic cosine of vector elements

Function	Function Prototype	Description
	ia,double *b,int ib,int n)	b[ib*j] = COSH(a[ia*j])
e_dvcub'	void e_dvcub(double *a,int ia,double *b,int ib,int n)	Cube vector elements b[ib*j] = a[ia*j]^3
e_dvdiv'	void e_dvdiv(double *a,int ia,double *b,int ib,double *c,int ic,int n)	Division of two vectors c[ic*j] = a[ia*j] / b[ib*j]
e_dveql'	void e_dveql(double *a,int ia,double *b,int ib,int *c,int n)	Evaluates if two complex vectors are equal c = AND(a[ia*j] == b[ib*j])
e_dvexp'	void e_dvexp(double *a,int ia,double *c, int ic,int n)	Vector exponential c[ic*j] = e^(a[ia*j])
e_dvexp10'	void e_dvexp10(double *a,int ia,double *c,int ic,int n)	Vector base 10 exponential c[ic*j] = 10^(a[ia*j])
e_dvexp2'	void e_dvexp2(double *a,int ia,double *c,int ic,int n)	Vector base 2 exponential c[ic*j] = 2^(a[ia*j])
e_dvfill'	void e_dvfill(double a, double *c,int ic,int n)	Vector fill c[ic*j] = a
e_dvfirst'	void e_dvfirst(double *a,int ia,int *c,int n)	Index of first non-zero vector element a[c] != 0 AND a[j] == 0 for j=0..(c-1)
e_dvfloat	void e_dvfloat(double *a, int ia, float *c, int ic, int n)	Convert from doubles to floats c[ic*j] = (float)a[ia*j]
e_dvgathr	void e_dvgathr(double *a,int *b,int ib,double *c,int ic,int n)	Gather c[ic*j] = a[b[ib*j]]
e_dvhypot'	void e_dvhypot(double *a,int ia,double *b,int ib,double *c,int ic,int n)	Elementwise square root of sum of squares of two vectors c[ic*j] = SQRT(a[ia*j]+b[ib*j])
e_dvifix'	void e_dvifix(double *a,int ia,int *c,int ic,int n,int f)	Conversion to integer by truncation and rounding if (f) c[ic*j] = (int)(a[ia*j]) else c[ic*j] = ROUND(a[ia*j])
e_dvklip'	void e_dvklip(double *a,int	Clip elements of vector so that values

Function	Function Prototype	Description
	ia,double b, double *c,int ic,int n)	are between +/- ABS(b) c[ic*j] = MAX(-ABS(b), MIN(ABS(b),a[ia*j]))
e_dvlast'	void e_dvlast(double *a,int ia,int *c,int n)	Index of last non-zero vector element a[c] != 0 AND a[j] == 0 for j=(c+1)..(n-1)
e_dvlim'	void e_dvlim(double *a,int ia,double b, double c,double *d,int id,int n)	Vector limit to single absolute value, the sign of which depends on whether the input exceeds a threshold if (a[ia*j] < b) d[id*j] = -c else d[id*j] = c
e_dvlint'	void e_dvlint(double *t,int nt,double *u, int ius,double *y,int iys, int n)	Vector linear interpolate y[iys*j] = t[nt] + (u[ius*j]-nt)*(t[nt+1]-t[nt]) where nt=(int)u[ius*j]
e_dvlog'	void e_dvlog(double *a,int ia,double *c, int ic,int n)	Natural logarithm of vector elements c[ic*j] = LN(a[ia*j])
e_dvlog10'	void e_dvlog10(double *a,int ia,double *c, int ic,int n)	Logarithm base 10 of vector elements c[ic*j] = LOG10(a[ia*j])
e_dvlog2'	void e_dvlog2(double *a,int ia,double *c, int ic,int n)	Logarithm base 2 of vector elements c[ic*j] = LOG2(a[ia*j])
e_dvma'	void e_dvma(double* a,int ia,double *b,int ib, double *c,int ic, double *d, int id,int n)	Vector multiplication and addition d[id*j] = (a[ia*j]*b[ib*j]) + c[ic*j]
e_dvmax'	void e_dvmax(double *a,int ia,double *b, int ib,double *c,int ic, int n)	Elementwise maximum of two vectors c[ic*j] = MAX(a[ia*j],b[ib*j])
e_dvmaxmg'	void e_dvmaxmg(double *a,int ia,double *b, int ib,double *c,int ic, int n)	Elementwise maximum magnitude of two vectors c[ic*j] = MAX(ABS(a[ia*j]),ABS(b[ib*j]))

Function	Function Prototype	Description
e_dvmin’	void e_dvmin(double *a,int ia,double *b, int ib,double *c,int ic,int n)	Elementwise minimum of two vectors c[ic*j] = MIN(a[ia*j],b[ib*j])
e_dvminmg’	void e_dvminmg(double *a,int ia,double *b, int ib,double *c,int ic, int n)	Elementwise minimum magnitude of two vectors c[ic*j] = MIN(ABS(a[ia*j]),ABS(b[ib*j]))
e_dvmma’	void e_dvmma(double *a,int ia,double *b, int ib,double *c,int ic, double *d,int id,double *e, int ie,int n)	Multiplication of two pairs of vectors and add results e[ie*j] = (a[ia*j]*b[ib*j]) + (c[ic*j]*d[id*j])
e_dvmmsb’	void e_dvmmsb(double *a,int ia,double *b, int ib,double *c,int ic,double *d,int id,double *e,int ie, int n)	Multiplication of two pairs of vectors and subtract results e[ie*j] = (a[ia*j]*b[ib*j]) - (c[ic*j]*d[id*j])
e_dvmov’	void e_dvmov(double *a,int ia,double *c, int ic,int n)	Move vector c[ic*j] = a[ia*j]
e_dvmrg’	void e_dvmrg(double *a,int ia,double *b, int ib,double *c,int ic,int m, int n)	Merge two sorted vectors into one sorted vector if (a[ia*i] < b[ib*j]) c[ic*k++] = a[ia*i++] else c[ic*k] = b[ib*j++]
e_dvmsa’	void e_dvmsa(double *a,int ia,double *b, int ib,double c,double *d, int id,int n)	Vector multiplication and scalar addition d[id*j] = a[ia*j]*b[ib*j] + c
e_dvmsb’	void e_dvmsb(double *a,int ia,double *b,int ib,double *c,int ic,double *d,int id,int n)	Vector multiplication and subtraction d[id*j] = (a[ia*j]*b[ib*j]) - c[ic*j]
e_dvmsn’	void e_dvmsn(double *a,int ia,double *b,int ib,double *c,int ic,double *d,int id,int n)	Subtract product of two vectors from a third vector d[id*j] = c[ic*j] - (a[ia*j]*b[ib*j])
e_dvmul’	void e_dvmul(double *a,int ia,double	Multiplication of two vectors

Function	Function Prototype	Description
	*b,int ib,double *c,int ic,int n)	c[ic*j] = a[ia*j] * b[ib*j]
e_dvnabs'	void e_dvnabs(double *a,int ia,double *c,int ic,int n)	Vector negative absolute value c[ic*j] = -ABS(a[ia*j])
e_dvneg'	void e_dvneg(double *a,int ia,double *c,int ic,int n)	Vector negate c[ic*j] = -a[ia*j]
e_dvneql'	void e_dvneql(double *a,int ia,double *b,int ib,int *c,int n)	Evaluates if two vectors are not equal c = OR(a[ia*j] != b[ib*j])
e_dvpolre	void e_dvpolrect (double *r,int ir,double *theta, int itheta, double *x,int ix,double *y,int iy,int n)	Convert from polar to rectangular coordinates c[j] = a[j]*cos(b[j]), d[j] = a[j]*sin(b[j])
e_dvpow'	void e_dvpow(double *a,int ia,double *b,int ib,double *c,int ic,int n)	Vector power c[ic*j] = a[ia*j]^b[ib*j]
e_dvprog'	void e_dvprog(double *a,int ia,double *c,int ic,int n)	Vector progression c[ic*j] = SUM(a[ia*i]) for i=0..j
e_dvramp'	void e_dvramp(double a,double b,double *c,int ic,int n)	Vector ramp c[ic*j] = a + j*b
e_dvrecip'	void e_dvrecip(double *a,int ia,double *c,int ic,int n)	Take the reciprocal of a vector c[ic*j] = 1 / a[ia*j]
e_dvrectp	void e_dvrectpol (double *x,int ix, double *y, int iy, double *r, int ir, double *theta, int itheta, int n)	Convert from rectangular to polar coordinates c[j] = SQRT(a[j]^2+b[j]^2), d[j] = ATAN(b[j] / a[j])
e_dvrfft'	void e_dvrfft(double *in,double *out,int n,int d)	Compute double real Fast Fourier Transform if (d) out = FFT(in) else out = IFFT(in) Packed into n/2 double complex values IM(out[0]) = Nyquist Point
e_dvrfftnd'	e_dvrfftnd(double *in,double	FFT similar to e_dvrfft that may or

Function	Function Prototype	Description
	*out,int n,int d)	may not scale results
e_drfftwts	int e_set_drfftwts(int n) int e_drfftwts(void)	Allocate double real FFT buffers Use in conjunction with e_dvrfft
e_dvrvrs'	void e_dvrvrs(double *c, int ic, int n)	Vector reverse inplace SWAP(c[ic*j], c[ic*(n-1-j)]) for j=0..floor(n/2)
e_dvsadd'	void e_dvsadd(double *a,int ia,double b,double *c,int ic,int n)	Addition of vector and scalar c[ic*j] = a[ia*j] + b
e_dvsam'	void e_dvsam(double *a,int ia,double b, double *c,int ic,double *d,int id,int n)	Vector scalar addition and multiplication d[id*j] = (a[ia*j]+b) * c[ic*j]
e_dvsbm'	void e_dvsbm(double *a,int ia,double *b, int ib,double *c,int ic,double *d, int id, int n)	Vector subtraction and multiplication d[id*j] = (a[ia*j]-b[ib*j]) * c[ic*j]
e_dvsbsbm'	void e_dvsbsbm(double *a,int ia,double *b, int ib,double *c,int ic, double *d,int id,double *e, int ie,int n)	Subtraction of two pairs of vectors and multiply the results e[ie*j] = (a[ia*j]-b[ib*j]) * (c[ic*j]-d[id*j])
e_dvsbsm'	void e_dvsbsm(double *a,int ia,double *b, int ib,double c,double *d, int id, int n)	Vector subtraction and scalar multiplication d[id*j] = (a[ia*j]-b[ib*j]) * c
e_dvscale'	void e_dvscale (double *a,int ia, long k, double *b,int ib,int n)	Scale a vector by a power of 2 b[ib*j] = a[ia*j] * 2^k
e_dvscatr	void e_dvscatr(double *a, int ia, int *b,int ib,double *c,int n)	Scatter c[b[ib*j]] = a[ia*j]
e_dvsdiv'	void e_dvsdiv(double *a,int ia,double b,double *c,int ic,int n)	Division of vector by scalar c[ic*j] = a[ia*j] / b
e_dvsfix	void e_dvsfix(double *a,int ia,short *c,int ic,int n)	Convert from doubles to shorts c[i*ic] =(short)a[i*ia]

Function	Function Prototype	Description
e_dvshrink'	void e_dvshrink(double *a,int ia,double *b, int ib, int n)	Average vector elements pairwise, cutting vector length in half b[ib*j] = 0.5 * (a[ia*(2*i)] + a[ia*(2*i+1)])
e_dvsin'	void e_dvsin(double *a,int ia,double *c, int ic,int n)	Sine of vector elements c[ic*j] = SIN(a[ia*j])
e_dvsinh'	void e_dvsinh (double *a,int ia,double *b, int ib,int n)	Hyperbolic sine of vector elements b[ib*j] = SINH(a[ia*j])
e_dvsma'	void e_dvsma(double *a,int ia,double b, double *c,int ic,double *d,int id,int n)	Vector scalar multiplication and addition d[id*j] = a[ia*j]*b + c[ic*j]
e_dvsmsa'	void e_dvsmsa(double *a,int ia,double b, double c,double *d,int id, int n	Vector scalar multiplication and scalar addition d[id*j] = a[ia*j]*b + c
e_dvsmsb'	void e_dvsmsb(double *a,int ia,double b, double *c,int ic,double *d, int id,int n)	Vector scalar multiplication and subtraction d[id*j] = a[ia*j]*b - c[ic*j]
e_dvsmul'	void e_dvsmul(double *a,int ia,double b, double *c,int ic,int n)	Multiplication of vector by scalar c[ic*j] = a[ia*j] * b
e_dvsort'	void e_dvsort(double *c, int n, int f)	Sort vector inplace in ascending (f==1) or descending order (f==-1)
e_dvsq'	void e_dvsq(double *a,int ia,double *c,int ic,int n)	Square vector elements c[ic*j] = a[ia*j]^2
e_dvsqrt'	void e_dvsqrt(double *a,int ia,double *c,int ic,int n)	Square root of vector elements c[ic*j] = a[ia*j]^2
e_dvss'	void e_dvss(double *a,int ia,double *b,int ib,double *c,int ic,double *d, int id, int n)	Subtract two complex vectors from one vector d[id*j] = a[ia*j] - b[ib*j] - c[ic*j]
e_dvssq'	void e_dvssq(double *a,int ia,double *c, int ic,int n)	Signed square vector elements c[ic*j] = ABS(a[ia*j])*a[ia*j]

Function	Function Prototype	Description
e_dvsub'	void e_dvsub(double *a,int ia,double *b, int ib,double *c,int ic,int n)	Subtraction of two vectors c[ic*j] = a[ia*j] - b[ib*j]
e_dvsubsq	void e_dvsubsq(double *a,int ia,double *b,int ib,double *c,int ic,int n)	Vector subtract and square result c[ic*j] = (a[ia*j] - b[ib*j])^2
e_dvswap'	void e_dvswap(double *a,int ia,double *b, int ib,int n)	Swap two vectors SWAP(a[ia*j], b[ib*j])
e_dvtan'	void e_dvtan(double *a,int ia,double *c, int ic,int n)	Tangent of vector elements c[ic*j] = TAN(a[ia*j])
e_dvtanh'	void e_dvtanh (double *a,int ia,double *b, int ib,int n)	Hyperbolic tangent of vector elements b[ib*j] = TANH(a[ia*j])
e_dvthr'	void e_dvthr(double *a, int ia, double b double *c, int ic, int n)	Vector clip to greater than scalar if (a[ia*j] < b) c[ic*j] = b else c[ic*j] = a[ia*j]
e_dvthres'	void e_dvthres(double *a,int ia,double b, double *c,int ic,int n)	Set vector elements below threshold to zero if (a[ia*j] < b) c[ic*j] = 0 else c[ic*j] = a[ia*j]
e_dvuafix	void e_dvafix(double *a,int ia,unsigned char *c,int ic,int n)	Convert from doubles to unsigned chars c[i*ic] =(unsigned char)a[i*ia]
e_dvuifix	void e_dvuifix(double *a,int ia,unsigned int *c,int ic,int n)	Convert from doubles to unsigned ints c[i*ic] =(unsigned int)a[i*ia]
e_dvusfix	void e_dvusfix(double *a,int ia,unsigned short *c,int ic,int n)	Convert from doubles to unsigned shorts c[i*ic] =(unsigned short)a[i*ia]

Integer Functions

Function	Function Prototype	Description
e_idotpr'	void e_idotpr(int *a,int ia,int *b,int ib, int *c,int n)	Integer dot product c = SUM(a[ia*j]*b[ib*j]
e_imaxmgv'	void e_imaxmgv(int *a,int ia,int *mv,int *mi, int n)	First maximum magnitude element and its C-style index mv = ABS(a[ia*mi]) = MAX(ABS(a[ia*j]))
e_imaxv'	void e_imaxv(int *a,int ia,int *c,int *ic, int n)	First maximum element and its C-style index c = a[ia*ic] = MAX(a[ia*j])
e_imaxv0'	void e_imaxv0(int *a,int ia,int *c,int n)	Maximum element c = MAX(a[ia*j])
e_imeamgv'	void e_imeamgv(int *a,int ia,float *c,int n)	Mean of vector elements' magnitudes c = SUM(ABS(a[ia*j])) / n
e_imeanv'	void e_imeanv(int *a,int ia,float *c,int n)	Mean of vector c = SUM(a[ia*j]) / n
e_imeasqv'	void e_imeasqv(int *a,int ia,float *c,int n)	Mean of vector elements squared c = SUM(a[ia*j]^2) / n
e_iminmgv'	void e_iminmgv(int *a,int ia,int *mv,int *mi, int n)	First minimum magnitude element and its C-style index mv = ABS(a[ia*mi]) = MIN(ABS(a[ia*j]))
e_iminv'	void e_iminv(int *a,int ia,int *c,int *ic,int n)	First minimum element and its C-style index c = a[ia*ic] = MIN(a[ia*j])
e_iminv0'	void e_iminv0(int *a,int ia,int *c,int n)	Minimum element c = MIN(a[ia*j])
e_immul'	void e_immul(int *a,int *b,int *c,int nrc, int ncc, int nca)	Integer matrix multiplication c[i][k] = SUM(a[i][j]*b[j][k]) for i=0..(nrc-1), j=0..(nca-1), k=0..(ncc-1)

Function	Function Prototype	Description
e_isvdiv'	void e_isvdiv(int a,int *b,int ib,int *c, int ic,int n)	Division of a scalar by a vector c[ic*j] = a / b[ib*j]
e_isve'	void e_isve(int *a,int ia,int *c,int n)	Sum of vector elements c = SUM(a[ia*j])
e_isvemg'	void e_isvemg(int *a,int ia,int *c,int n)	Sum of vector elements'magnitudes c = SUM(ABS(a[ia*j]))
e_isvesq'	void e_isvesq(int *a,int ia,int *c,int n)	Sum of vector elements squared c = SUM(a[ia*j]^2)
e_isvessq'	void e_isvessq(int *a,int ia,int *c,int n)	Sum of vector elements sign-squared c = SUM(ABS(a[ia*j]) * a[ia*j])
e_isvsub'	void e_isvsub(int a,int *b,int ib,int *c, int ic, int n)	Subtract a vector from a scalar c[ic*j] = a - b[ib*j]
e_ivaa'	void e_ivaa(int *a,int ia,int *b,int ib,int *c,int ic,int *d,int id,int n)	Addition of three integer vectors d[id*j] = a[ia*j] + b[ib*j] + c[ic*j]
e_ivabs'	void e_ivabs(int *a,int ia,int *c,int ic,int n)	Absolute value of vector elements c[ic*j] = ABS(a[ia*j])
e_ivadd'	void e_ivadd(int *a,int ia,int *b,int ib, int *c, int ic, int n)	Addition of two vectors c[ic*j] = a[ia*j] + b[ib*j]
e_ivclip'	void e_ivclip(int *a,int ia,int b,int c,int *d,int id,int n)	Clip elements of an integer vector so values are between min and max d[id*j] = MAX(b,MIN(c,a[ia*j]))
e_ivdiv'	void e_ivdiv(int *a,int ia,int *b,int ib, int *c, int ic, int n)	Division of two vectors c[ic*j] = a[ia*j] / b[ib*j]
e_ivfill'	void e_ivfill(int a, int *c,int ic,int n)	Integer vector fill c[ic*j] = a
e_ivfloat'	void e_ivfloat(int *a,int ia,float *c,int ic, int n)	Convert integers to floats c[ic*j] = (float)a[ia*j]
e_ivlim'	void e_ivlim(int *a,int ia,int b,int c,int *d, int id, int n)	Vector limit to single absolute value, the sign of which depends on whether the input exceeds a threshold

Function	Function Prototype	Description
		if (a[ia*j] < b) d[id*j] = -c else d[id*j] = c
e_ivlog10'	void e_ivlog10(int *a,int ia,int *c,int ic, int n)	Logarithm base 10 of vector elements c[ic*j] = LOG10(a[ia*j])
e_ivlog2'	void e_ivlog2(int *a,int ia,int *c,int ic, int n)	Logarithm base 2 of vector elements c[ic*j] = LOG2(a[ia*j])
e_ivmax'	void e_ivmax(int *a,int ia,int *b,int ib, int *c,int ic,int n)	Elementwise maximum of two vectors c[ic*j] = MAX(a[ia*j],b[ib*j])
e_ivmaxmg'	void e_ivmaxmg(int *a,int ia,int *b,int ib, int *c,int ic,int n)	Elementwise maximum magnitude of two vectors c[ic*j] = MAX(ABS(a[ia*j]),ABS(b[ib*j]))
e_ivmin'	void e_ivmin(int *a,int ia,int *b,int ib, int *c,int ic,int n)	Elementwise minimum of two vectors c[ic*j] = MIN(a[ia*j],b[ib*j])
e_ivminmg'	void e_ivminmg(int *a,int ia,int *b,int ib, int *c,int ic,int n)	Elementwise minimum magnitude of two vectors c[ic*j] = MIN(ABS(a[ia*j]),ABS(b[ib*j]))
e_ivmod'	void e_ivmod(int *a,int ia,int *b,int ib, int *c,int ic,int n)	Modulus, integer division remainder c[ic*j] = a[ia*j] % b[ib*j]
e_ivmul'	void e_ivmul(int *a,int ia,int *b,int ib, int *c,int ic,int n)	Multiplication of two vectors c[ic*j] = a[ia*j] * b[ib*j]
e_ivneg'	void e_ivneg(int *a,int ia,int *c,int ic, int n)	Vector negate c[ic*j] = -a[ia*j]
e_ivsadd'	void e_ivsadd(int *a,int ia,int b,int *c, int ic,int n)	Addition of vector and scalar c[ic*j] = a[ia*j] + b
e_ivsdiv'	void e_ivsdiv(int *a,int ia,int b,int *c, int ic,int n)	Division of vector by scalar c[ic*j] = a[ia*j] / b
e_ivsma'	void e_ivsma(int *a,int ia,int b,int *c,int ic, int *d,int id,int n)	Vector scalar multiplication and addition d[id*j] = a[ia*j]*b + c[ic*j]

Function	Function Prototype	Description
e_ivsmul'	void e_ivsmul(int *a,int ia,int b,int *c, int ic,int n)	Multiplication of vector by scalar c[ic*j] = a[ia*j] * b
e_ivsq'	void e_ivsq(int *a,int ia,int *c,int ic,int n)	Square vector elements c[ic*j] = a[ia*j]^2
e_ivssq'	void e_ivssq(int *a,int ia,int *c,int ic,int n)	Signed square vector elements c[ic*j] = ABS(a[ia*j])*a[ia*j]
e_ivsub'	void e_ivsub(int *a,int ia,int *b,int ib, int *c,int ic,int n)	Subtraction of two vectors c[ic*j] = a[ia*j] - b[ib*j]
e_ivthr'	void e_ivthr(int *a,int ia,int b,int *c,int ic,int n)	Vector clip to greater than scalar if (a[ia*j] < b) c[ic*j] = b else c[ic*j] = a[ia*j]
e_ivthres'	void e_ivthres(int *a,int ia,int b,int *c, int ic,int n)	Set vector elements below threshold to zero if (a[ia*j] < b) c[ic*j] = 0 else c[ic*j] = a[ia*j]

Split Complex Functions

Function	Function Prototype	Description
e_zdotc'	e_zdotc(float *a_re, float *a_im, int ia, float *b_re, float *b_im, int ib, float *c_re, float *c_im, int n)	Complex dot product of first vector with conjugate of second (c_re,c_im)=SUM((a_re[j*ia],-a_im[j*ia])*(b_re[j*ib],b_im[j*ib]))
e_zdotpr'	e_zdotpr(float *a_re, float *a_im, int ia, float *b_re, float *b_im, int ib, float *c_re, float *c_im, int n)	Complex dot product of two vectors (c_re,c_im)=SUM((a_re[j*ia], a_im[j*ia])*(b_re[j*ib],b_im[j*ib]))
e_zfird'	e_zfird(float *a_re, float *a_im, float *b_re, float *b_im, int nb, float *c_re, float *c_im, int d, int n)	FIR filter with decimation (c_re,c_im)=SUM((a_re[i*d+j],a_im[i*d +j])*(b_re[j],b_im[j])) for i=0..(n-1), j=0..(nb-1)
e_zmmul'	e_zmmul(float *a_re, float *a_im, float *b_re, float *b_im, float *c_re, float *c_im, int nrc, int ncc, int nca)	Complex matrix multiply (c_re[i][k],c_im[i][k])=SUM((a_re[i][j],a_im[i][j])*(b_re[j][k],b_im[j][k])) for i=0..(nrc-1), j=0..(nca-1), k=0..(ncc-1)
e_zmtran'	e_zmtran(float *aa_re, float *aa_im, float *cc_re, float *cc_im, int n, int m, int t)	Complex matrix transpose (not inplace) if (t=0) transpose else conjugate transpose (hermitian)
e_zmtran2'	e_zmtran2(float *in_re, float *in_im, float *out_re, float *out_im, int ncin, int nrin, int tcin, int trin)	Complex matrix transpose of submatrix (not inplace) out_re[i][j]=in_re[i][j] out_im[i][j]=in_im[i][j] for j=0..(nrin-1), i=0..(ncin-1) where submatrix is size nrin x ncin

Function	Function Prototype	Description
		and matrix is size trin x tcin
e_zpolar'	e_zpolar(float *a_re, float *a_im, int ia, float *c_re, float *c_im, int ic, int n)	Convert rectangular to polar coordinates c_re[ic*j]=ABS((a_re[ia*j],a_im[ia*j])) c_im[ic*j]=ATAN(a_im[ia*j] / a_re[ia*j])
e_zrect'	e_zrect(float *a_re, float *a_im, int ia, float *c_re, float *c_im, int ic, int n)	Convert polar to rectangular coordinates c_re[ic*j]=a_re[ia*j]*cos(a_im[ia*j]) c_im[ic*j]=a_re[ia*j]*sin(a_im[ia*j])
e_zrvadd'	e_zrvadd(float *a_re, float *a_im, int ia, float *b, int ib, float *c_re, float *c_im, int ic, int n)	Add real vector b to complex vector a c_re[ic*j]=a_re[ia*j]+b[ib*j] c_im[ic*j]=a_im[ia*j]
e_zrvdiv'	e_zrvdiv(float *a_re, float *a_im, int ia, float *b, int ib, float *c_re, float *c_im, int ic, int n)	Divide complex vector a by real vector b c_re[ic*j]=a_re[ia*j]/b[ib*j] c_im[ic*j]=a_im[ia*j]/b[ib*j]
e_zrvmul'	e_zrvmul(float *a_re, float *a_im, int ia, float *b, int ib, float *c_re, float *c_im, int ic, int n)	Multiply complex vector a by real vector b c_re[ic*j]=a_re[ia*j]*b[ib*j] c_im[ic*j]=a_im[ia*j]*b[ib*j]
e_zrvsub'	e_zrvsub(float *a_re, float *a_im, int ia, float *b, int ib, float *c_re, float *c_im, int ic, int n)	Subtract real vector b from complex vector a c_re[ic*j]=a_re[ia*j]-b[ib*j] c_im[ic*j]=a_im[ia*j]
e_zsvemg'	e_zsvemg(float *a_re, float *a_im, int ia, float *c, int n)	Sum of vector element magnitudes c=SUM(ABS((a_re[j*ia],a_im[j*ia])))
e_zsvemgs'	e_zsvemgs(float *a_re, float *a_im, int ia, float *c, int n)	Sum of vector element magnitudes squared c=SUM(ABS((a_re[j*ia],a_im[j*ia]))^2)

Function	Function Prototype	Description
e_zsvmul’	e_zsvmul(float *a_re, float *a_im, float *b_re, float *b_im, int ib, float *c_re, float *c_im, int ic, int n)	Complex scalar multiply (c_re[ic*j],c_im[ic*j])=(a_re,a_im)*(b_re[ib*j],b_im[ib*j])
e_zvabs’	e_zvabs(float *a_re, float *a_im, int ia, float *c, int ic, int n)	Magnitude c[ic*j]=ABS((a_re[ia*j],a_im[ia*j]))
e_zvadd’	e_zvadd(float *a_re, float *a_im, int ia, float *b_re, float *b_im, int ib, float *c_re, float *c_im, int ic, int n)	Addition of two complex vectors c_re[ic*j]=a_re[ia*j]+b_re[ib*j] c_im[ic*j]=a_im[ia*j]+b_im[ib*j]
e_zvasub’	e_zvasub(float *a_re, float *a_im, int ia, float *b_re, float *b_im, int ib, float *c_re, float *c_im, int ic, float *d_re, float *d_im, int id, int n)	Add two complex vectors and subtract away another vector d_re[id*j]=a_re[ia*j]+b_re[ib*j]-c_re[ic*j] d_im[id*j]=a_im[ia*j]+b_im[ib*j]-c_im[ic*j]
e_zvclip’	e_zvclip (float *a_re, float *a_im, int ia, float min_re, float min_im, float max_re, float max_im, float *out_re, float *out_im, int iout, int n)	Clip values so they are between (min_re,min_im) and (max_re,max_im) out_re[iout*j]=MAX(min,MIN(max,a_re[ia*j])) out_im[iout*j]=MAX(min,MIN(max,a_im[ia*j]))
e_zvclr’	e_zvclr(float *c_re, float *c_im, int ic, int n)	Clear to zero c_re[ic*j]=0 c_im[ic*j]=0
e_zvconj’	e_zvconj(float *a_re, float *a_im, int ia, float *c_re, float *c_im, int ic, int n)	Complex conjugate c_re[ic*j]=a_re[ia*j] c_im[ic*j]=-a_im[ia*j]
e_zvcub’	e_zvcub(float *a_re, float *a_im, int ia, float *b_re, float *b_im, int ib, int n)	Cube (b_re[ib*j],b_im[ib*j])=(a_re[ia*j],a_im[ia*j])^3

Function	Function Prototype	Description
e_zvdiv’	e_zvdiv(float *a_re, float *a_im, int ia, float *b_re, float *b_im, int ib, float *c_re, float *c_im, int ic, int n)	Divide (c_re[ic*j],c_im[ic*j])=(a_re[ia*j],a_im[ia*j]) / (b_re[ib*j],b_im[ib*j])
e_zveql’	e_zveql(float *a_re, float *a_im, int ia, float *b_re, float *b_im, int ib, int *c, int n)	Evaluates if two complex vectors are equal c = AND(a_re[ia*j]==b_re[ib*j] && a_im[ia*j]==b_im[ib*j])
e_zvexp’	e_zvexp(float *a, int ia, float *c_re, float *c_im, int ic, int n)	Complex exponent (c_re[ic*j],c_im[ic*j])=e^(i*a[j*ia])
e_zvfftb’	e_zvfftb(float *in_re, float *in_im, float *out_re, float *out_im, int n, int d)	Compute Fast Fourier Transform if (d) (out_re,out_im) = FFT((in_re,in_im)) else (out_re,out_im) = IFFT((in_re,in_im))
e_zvfill’	e_zvfill(float a_re, float a_im, float *c_re, float *c_im, int ic, int n)	Vector fill c_re[ic*j] = a_re c_im[ic*j] = a_im
e_zvifftbnd’	e_zvifftbnd(float *in_re, float *in_im, float *out_re, float *out_im, int n)	Inverse FFT that may or may not scale results
e_zvmags’	e_zvmags(float *a_re, float *a_im, int ia, float *c, int ic, int n)	Magnitude squared c[ic*j]=ABS((a_re[ia*j],a_im[ia*j]))^2
e_zvmov’	e_zvmov(float *a_re, float *a_im, int ia, float *c_re, float *c_im, int ic, int n)	Copy c_re[ic*j]=a_re[ia*j] c_im[ic*j]=a_im[ia*j]
e_zvmsn’	e_zvmsn(float *a_re, float *a_im, int ia, float *b_re, float *b_im, int ib, float *c_re, float *c_im, int ic, float *d_re, float *d_im,	Subtract product of two vectors from a third vector (d_re[id*j],d_im[id*j])=(c_re[ic*j],c_im[ic*j])-

Function	Function Prototype	Description
	int id, int n)	(a_re[ia*j],a_im[ia*j])*(b_re[ib*j],b_im[ib*j])

e_zvmul'	e_zvmul(float *a_re, float *a_im, int ia, float *b_re, float *b_im, int ib, float *c_re, float *c_im, int ic, int n, int f)	Multiplication of two complex vectors
		If (f==-1)
		(c_re[ic*j],c_im[ic*j])=(a_re[ia*j],-a_im[ia*j]) * (b_re[ib*j],b_im[ib*j])
		else
		(c_re[ic*j],c_im[ic*j])=(a_re[ia*j],a_im[ia*j]) * (b_re[ib*j],b_im[ib*j])

e_zvneg'	e_zvneg(float *a_re, float *a_im, int ia, float *c_re, float *c_im, int ic, int n)	Negate c_re[ic*j]=-a_re[ia*j] c_im[ic*j]=-a_im[ia*j]
e_zvneql'	e_zvneql(float *a_re, float *a_im, int ia, float *b_re, float *b_im, int ib, int *c, int n)	Evaluates if two complex vectors are not equal c = OR(a_re[ia*j]!=b_re[ib*j] || a_im[ia*j]!=b_im[ib*j])
e_zvphas'	e_zvphas(float *a_re, float *a_im, int ia, float *c, int ic, int n)	Find the phase of the complex vector c[ic*j]=atan(a_im[ia*j] / a_re[ia*j])
e_zvramp'	e_zvramp(float a_re, float a_im, float b_re, float b_im, float *c_re, float *c_im, int ic, int n)	Complex vector ramp c_re[ic*j]=a_re+j*b_re c_im[ic*j]=a_im+j*b_im
e_zvrcip'	e_zvrcip(float *a_re, float *a_im, int ia, float *c_re, float *c_im, int ic, int n)	Reciprocal (c_re[ic*j],c_im[ic*j])=1/(a_re[ia*j],a_im[ia*j])
e_zvrsmul'	e_zvrsmul(float *a_re, float *a_im, int ia, float b, float *c_re, float *c_im, int ic, int n)	Multiplication of complex vector by real scalar c_re[ic*j]=a_re[ia*j]*b c_im[ic*j]=a_im[ia*j]*b
e_zvsma'	e_zvsma(float *a_re, float *a_im, int ia, float *b_re, float *b_im, float *c_re, float *c_im, int ic,	Scalar multiplication and add y_re[iy*j]=a_re[ia*j]*b+c_re[ic*j] y_im[iy*j]=a_im[ia*j]*b+c_im[ic*j]

Function	Function Prototype	Description
	float *y_re, float *y_im, int iy, int n)	
e_zvss’	e_zvss(float *a_re, float *a_im, int ia, float *b_re, float *b_im, int ib, float *c_re, float *c_im, int ic, float *d_re, float *d_im, int id, int n)	Subtract two complex vectors from one vector d_re[id*j]=a_re[ia*j]-b_re[ib*j]-c_re[ic*j] d_im[id*j]=a_im[ia*j]-b_im[ib*j]-c_im[ic*j]
e_zvsub’	e_zvsub(float *a_re, float *a_im, int ia, float *b_re, float *b_im, int ib, float *c_re, float *c_im, int ic, int n)	Subtraction of two complex vectors c_re[ic*j]=a_re[ia*j]-b_re[ib*j] c_im[ic*j]=a_im[ia*j]-b_im[ib*j]