FASTSERIES

FASTSERIES LIBRARY
USER'S MANUAL

FAST CAPTURE
FAST PROCESSING
FAST RESULTS

FASTSERIES PCI BOARD

FastVision
FastImage 1300
FastFrame 1300

FAST SERIES PMC

FastMem
Fast4 1300
Fast I/O 1300

30002-00150

COPYRIGHT NOTICE

Copyright ã 2002 by Alacron Inc.

Alacron makes no warranty for the use of its products, assumes no responsibility for any error, which may appear in this document, and makes no commitment to update the information contained herein. Alacron Inc. retains the right to make changes to this manual at any time without notice.

Document Name: FastSeries Library User's Manual

Document Number: 30002-00150

Revision History: 1.6 June 3, 2002

Trademarks:

Alacronâ is a registered trademark of Alacron Inc.
AltiVecÔ is a trademark of Motorola Inc.
Channel LinkÔ is a trademark of National Semiconductor
CodeWarriorâ is a registered trademark of Metrowerks Corp.
FastChannelâ is a registered trademark of Alacron Inc.
FastSeriesâ is a registered trademark of Alacron Inc.
Fast4â**, FastFrame 1300**â**, FastImage**â**, FastI/O**â**, and FastVision**® are registered trademarks of Alacron Inc.
FireWireÔ is a registered trademark of Apple Computer Inc.
3MÔ is a trademark of 3M Company
MS DOSâ is a registered trademark of Microsoft Corporation
SelectRAMÔ is a trademark of Xilinx Inc.
SolarisÔ is a trademark of Sun Microsystems Inc.
TriMediaÔ is a trademark of Philips Electronics North America Corp.
Unixâ is a registered trademark of Sun Microsystems Inc.
VirtexÔ is a trademark of Xilinx Inc.
WindowsÔ**, Windows 95**Ô**, Windows 98**Ô**, Windows 2000**Ô**, and Windows NT**Ô are trademarks of Microsoft

All trademarks are the property of their respective holders.

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Telephone: 603-891-2750
Fax: 603-891-2745

Web Site:
http://www.alacron.com/

Email:
sales@alacron.com, or support@alacron.com

TABLE OF CONTENTS

MANUAL FIGURES & TABLES

FIGURE	PAGE	SUBJECT	TABLE	PAGE	SUBJECT

OTHER ALACRON MANUALS

Alacron manuals cover all aspects of FastSeries hardware and software installation and operation. Call Alacron at 603-891-2750 and ask for the appropriate manuals from the list below if they did not come in your FastSeries shipment.

- 30002-00146 FastImage and FastFrame HW Installation Manual
- 30002-00148 ALFAST Runtime Software Programmer's Guide & Reference
- 30002-00150 FastSeries Library User's Manual
- 30002-00153 Fast I/O Hardware User's Manual
- 30002-00155 FastMem Hardware User's Manual
- 30002-00162 FOIL – **F**astSeries **O**bject **I**maging **L**ibrary User's Manual
- 30002-00169 ALRT Runtime Software Programmer's Guide & Reference
- 30002-00170 ALRT, ALFAST & FASTLIB Software Installation Manual for Linux
- 30002-00171 ALRT, ALFAST, & FASTLIB Software Installation for Windows NT
- 30002-00173 FastMem Programmer's Guide & Reference
- 30002-00174 FastMem Hardware Installation Manual
- 30002-00176 FastImage 1300 Hardware User's Manual
- 30002-00180 Fast4 1300 Hardware User's Manual
- 30002-00184 FastSeries Getting Started Manual
- 30002-00183 FastImage 1300 Camera Integration User's Manual
- 30002-00185 FastVision Hardware User's Manual
- 30002-00186 FastVision Software User's Manual
- 30002-00187 FastFrame 1300 Hardware User's Manual

I. INTRODUCTION

A. Purpose

The *FastSeries Library User's Manual* provides calling specifications and descriptions for the Alacron FastSeries Library of vector and image processing functions.

B. Audience

This manual is intended for technical personnel responsible for developing application software to run on Alacron boards. This manual assumes familiarity with operating system commands to configure the software and with the C programming language.

II. FASTSERIES LIBRARY

The Alacron FastSeries Library for the FastSeries family of TriMedia-based processor boards is based on Alacron's Vector Library (VLIB) and Real-Time Image-Processing Library (RIPL). The routine names and calling sequences are largely unchanged from the earlier library versions.

The Alacron FastSeries supports Alacron's FastSeries family of Processor Boards and Processing Daughter cards

- 1-D vector functions used in digital signal processing and graphics.
- 2-D image processing functions that operate on 8-bit unsigned integer, 16-bit unsigned integer, and 32-bit floating point image data.

This section provides an overview of the FastSeries Library data types and functions. Function calling sequences, return values and other specifics are provided in the next chapter.

A. Host-Linked and TriMedia-Linked Library Versions

The FastSeries Library is distributed with two versions, one that links to a TriMedia program and another that links instead to the Host (Pentium) program.

TriMedia programs that wish to use the FastSeries Library should link to **libfastlib.a**.

Host (Pentium) programs that wish to access the FastSeries Library functions should link to **libfastlib.lib**.

B. Include File

Application programs using the FastSeries Library should include **<fastlib.h>,** which is in the **%FASTLIB%\include** installation directory.

C. Data Types

1. Scalar Data Types

a) 8-bit integer (`unsigned char`)

8-bit integer data types are defined as one byte unsigned integer elements that range from 0 to 255.

b) 16 bit integer (`unsigned short`)

16 bit integer data types are defined as two byte unsigned integer elements that range from 0 to 65535.

c) Float (`float`)

Float data types are defined as IEEE standard 32 bit single precision floating point numbers.

d) Packed Binary Least-to-most (`l2m`)

Packed binary l2m are binary images packed 8 binary pixels to each byte, ordered from least significant bit to most significant bit.

```
d0 - pixel n
d1 - pixel n+1
.
.
d7 - pixel n+7
```

e) Packed Binary Most-to-least (`m2l`)

Packed binary m2l are binary images packed 8 binary pixels to each byte, ordered from most significant bit to least significant bit.

```
d0 - pixel n+7
d1 - pixel n+6
.
.
d7 - pixel n
```

2. Vectors

A vector is a one-dimensional array of values. Its elements occupy successive locations in memory. A *vector reference* in the algorithms has the form:

a[i]

Where the integer index *i* range from 0 to *n*-1 (*n* being the total number of elements in the vector). A vector can be made of integer, real, or complex values.

a) Vector Strides

Many FastSeries Library functions contain vector stride arguments to specify which elements of a vector are to be processed. Vector strides allow the functions to handle individual rows of matrices or operate on complex vectors as real vectors.

b) Element Stride

Vector functions use an *element stride* for one or more argument vectors, shown as argument name *ia* for the stride of argument *a*, *ib* for the stride of argument *b*, and so on.

An element stride of *k* specifies that every *k*th real element of a vector (physical indexes 0, *k*, 2*k*,...) is to be processed. The element stride refers to the spacing of the real values in the vector array. To reference every element of a vector of real values, use a stride of 1. To reference every element in a vector of complex values (or every other element in a real vector), use a stride of 2.

c) Row Stride

The *row stride* allows a function to handle a sub-matrix within a full matrix. The row stride parameter specifies the number of full matrix elements between successive elements in a column of the sub-matrix.

For example, consider a 50x50 sub-matrix within a 100x100 full matrix. The distance between the first element in row 0 and the first element in row 1 of the sub-matrix is 100 (the row size of the full matrix). The row stride for this sub-matrix would be 100. The row stride is different from the sub-matrix *row size*, which would be 50 in this example.

3. Matrices and Images

A matrix or image is a two-dimensional array of values defined by a pointer (or array address), a vertical stride, the number of rows and the number of columns. These values may either be real or complex. The vertical stride defines the address increment from one row of the array to the next, and allows the referencing of “sub-arrays” of the image.

The FastSeries Library uses row-major order. Successive locations in memory contain successive elements of a row, until the end of the row, which is followed by the first element of the next row. If a program desires column-major order, swap the row and column input arguments to achieve the desired result.

4. Complex Scalars, Vectors, and Matrices

Some routines use type **complex** scalars, vectors, or matrices. A complex value in the FastSeries Library is a pointer to an array of floats:

float *a

The length of the array determines the “type” of the complex value. A complex scalar has length 2, one real value and one imaginary value. A complex vector has length 2*N for N complex vector elements. A complex matrix or image has 2*N*M for M rows of 2*N complex elements per row. The vector and matrix values use a corresponding stride to specify the values for M and N.

D. Summary of Functions

The FastSeries Library contains the following functions:

Function	Description
AbsDiff	Absolute value of difference
AbsSum	Sum of absolute values
Add	Add images
AddScalar	Add scalar to image
And16	Logical and of 16 bit image
And8	Logical and of 8-bit image
AndNot16	Logical AND 16 bit image with complement
AndNot8	Logical AND 8-bit image with complement
ve	Accumulate average of images
Ave16	Accumulate average of 16 bit image
Ave8	Accumulate average of 8-bit image
bldwts	build FFT weights
blkman	blackman window multiply
cdotpr	complex dot product
cfft	complex FFT (in-place)
cfft2d	complex 2D FFT (in-place)
cfftb	complex FFT (not-in-place)
cfftsc	complex FFT scale
cfftss	complex FFT scale with stride
cimul	complex image multiplication
conv	linear convolution
conv	convolution and correlation
Conv2d	general 2D convolution
Conv3x3	3x3 convolution
Conv5x5	5x5 convolution
Conv7x7	7x7 convolution
ConvFFT	convolution via FFT
cvadd	complex vector add
cvcomb	complex vector combine
cvconj	complex vector conjugate
cvcsml	complex vector complex scalar multiply
cvdiv	complex vector divide
cvexp	complex vector exponential
cvmags	complex vector magnitude squared
cvmov	complex vector move
cvmul	complex vector multiply
cvneg	complex vector negate
cvrcip	complex vector reciprocal
cvsma	complex vector scalar multiply and add

cvsub	complex vector subtract
Div	Divide images
dotpr	dot product
fastlibversion	Return FastSeries Library version string
Fill	Fill float image with constant
Fill16	Fill 16 bit image with constant
Fill8	Fill 8-bit image with constant
Fix16	Convert float image to 16 bit integer
Fix8	convert float image to 8-bit integer
fixinta	fix pixels in format A (high/low)
fixintb	fix pixels in format B (low/high)
Float16	Convert 16 bit integer image to float
Float8	Convert 8-bit integer image to float
fltinta	float pixels in format A (high/low)
fltintb	float pixels in format B (low/high)
hamm	hamming window multiply
hann	hanning window multiply
hist	histogram
HistoEqual10	Histogram Equalization on 10 bit image
HistoEqual12	Histogram Equalization on 12 bit image
HistoEqual14	Histogram Equalization on 14 bit image
HistoEqual16	Histogram Equalization on 16 bit image
HistoEqual8	Histogram Equalization on 8-bit image
Histogram	Calculate histogram of float image
Histogram10	Calculate histogram for 10 bit image
Histogram12	Calculate histogram for 12 bit image
Histogram14	Calculate histogram for 14 bit image
Histogram16	Calculate histogram for 16 bit image
Histogram8	Calculate histogram for 8-bit image
Ifree	Free dynamically allocated image
Imalloc	Dynamically allocate a float image
Imalloc16	Dynamically allocate a 16-bit image
Imalloc8	Dynamically allocate an 8-bit image
Interlace	Convert non-interlaced image to interlaced image
Kirsch	Kirsch operator
Lut8	Perform lookup on 8-bit image to 8-bit image
Lut8f	Perform lookup on 8-bit image to float image
Lut8s	Perform lookup on 8-bit image to 16 bit image
Lutf	Perform lookup on float image to float image
lveq	logical vector equal
lvge	logical vector greater than or equal
lvgt	logical vector greater than
lvne	logical vector not equal
lvnot	logical vector not
matinv	matrix inverse
Max	Maximum of images
maxmgv	maximum magnitude of a vector
maxv	maximum element of a vector
meamgv	mean of vector element magnitudes
measqv	mean of vector element squares
Median	Median filter

Min	Minimum of images
minmgv	minimum magnitude element of a vector
minv	minimum element of a vector
mmul	real matrix multiply
Mov	Move float image
Mov16	Move 16 bit image
Mov8	Move 8-bit image
mtrans	matrix transpose
MulScalar	Multiply image by scalar
Not16	Compute complement of 16 bit image
Not8	Compute complement of 8-bit image
Or16	Logical OR 16 bit image
Or8	Logical OR 8-bit image
polar	rectangular to polar conversion
PowerSpectrum	power spectrum
Prewitt	Prewitt operator
rect	polar to rectangular conversion
Reflect	Reflect float image (vertical, horizontal, diagonal)
Reflect16	Reflect 16 bit image (vertical, horizontal, diagonal)
Reflect8	Reflect 8-bit image (vertical, horizontal, diagonal)
RegLut8 [1]	Allocate 8-Bit Lookup Table
RegLut8s [1]	Allocate 16-Bit Lookup Table
rfft	real to complex FFT (in-place)
Rfft2d	real 2d FFT
rfft2d	real to complex 2D FFT (in-place)
rfftb	real to complex FFT (not-in-place)
rfftsc	real FFT scale and format
rmsqv	root mean square of vector elements
Roberts	Roberts operator
Rotate	Rotate float image
Rotate16	Rotate 16 bit image
Rotate8	Rotate 8-bit image
Sobel	Sobel operator
Sub	Subtract images
Sum	Sum float image to float image
Sum16f	Sum 16 bit image to float image
Sum16i	Sum 16 bit image to 32 bit integer
Sum8f	Sum 8-bit image to float image
Sum8i	Sum 8-bit image to 32 bit image
svdiv	scalar vector divide
sve	sum of vector elements
svemg	sum of vector element magnitudes
svesq	sum of vector element squares
svs	sum of vector signed element squares
Thr8	Threshold 8-bit image to 8-bit image
Thr8l2m	Threshold 8-bit image to packed binary l2m (lsb to msb)
Thr8m2l	Threshold 8-bit image to packed binary m2l (msb to lsb)
Uninterlace	Convert interlaced image to non-interlaced image

uprfft2	unpack results of rfft2d
vabs	vector absolute value
vadd	vector add
valog10	vector anti-log base 10
vam	vector add and multiply
vand	vector logical AND
vatan	vector arctangent
vatan2	vector two argument arctangent
vclip	vector clip
vclr	vector clear
vcos	vector cosine
vdist	vector distance
vdiv	vector divide
venvlp	vector envelope
veqv	vector logical EQUIVALENCE
vexp	vector exponential
vfill	vector fill with constant
vfix	convert a float vector to 32-bit integer
vfix	vector fix to integer
vfix16	convert a float vector to 16-bit integer
vfix16	vector fix to short integer
vfix32	convert a float vector to 32-bit long
vfix32	vector fix to long integer
vfix8	convert a float vector to 8-bit integer
vfix8	vector fix to 8 bit integer
vflt	convert a 32-bit integer vector to float
vflt	vector float
vflt16	convert a 16-bit signed integer vector to float
vflt16	vector float short integers
vflt32	convert a 32-bit signed integer vector to float
vflt32	vector float long integers
vflt8	convert an 8-bit singed integer vector to float
vflt8	vector float byte integers
vfltu16	convert a 16-bit unsigned integer vector to float
vfltu16	vector float unsigned short integers
vfltu8	convert an 8-bit unsigned integer vector to float
vfltu8	vector float unsigned byte integers
vfrac	vector truncate to fraction
vgathr	vector gather
vimag	extract imaginaries of complex vector
vlim	vector limit
vlog	vector natural logarithm
vlog10	vector base 10 logarithm
vma	vector multiply and add
vmax	vector maximum of two vectors
vmaxmg	vector maximum magnitude of two vectors
vmin	vector minimum of two vectors
vminmg	vector minimum magnitude of two vectors
vmov	vector move
vmsa	vector multiply and scalar add
vmsb	vector multiply and subtract
vmul	vector multiply
vnabs	vector negative absolute value
vneg	vector negate
vor	vector logical OR
vpoly	vector polynomial evaluation

vramp	vector fill with ramp
vrcip	vector reciprocal
vreal	extract reals of complex vector
vsadd	vector scalar add
vsbm	vector subtract and multiply
vscal	vector scale and fix
vscatr	vector scatter
vsdiv	vector scalar divide
vsin	vector sine
vsma	vector scalar multiply and add
vsmb	vector scalar multiply and subtract
vsmsa	vector scalar multiply and scalar add
vsmsb	vector scalar multiply and subtract
vsmul	vector scalar multiply
vsq	vector square
vsqrt	vector square root
vssq	vector signed square
vsub	vector subtract
vswap	vector swap
vtan	vector tangent
vthr	alias for vector threshold
vthresh	vector threshold
Xgradient	X gradient filter
Xor16	Logical XOR 16 bit image
Xor8	Logical XOR 8-bit image
Ygradient	Y gradient filter
Zoom	Zoom float image
Zoom16	Zoom 16 bit image
Zoom8	Zoom 8-bit image

1. Standard Calling Sequence

The FastSeries Library uses a standard two or three operand calling sequence for unary and binary operators; each function has one or more input image data arguments and usually one output image argument. The standard two operand calling sequence is:

***fctn8* (unsigned char a[], int ia, unsigned char c[], int ic, int nr, int nc);**

The standard three operand calling sequence is:

***fctn* (float a[], int ia, float b[], int ib, float c[], int ic, int nr, int nc);**

In these cases, the arguments ***a[]*** and ***b[]*** reference input arguments, ***c[]*** references the output argument, ***nr*** and ***nc*** reference the number of rows and columns to be operated on. The arguments ***ia, ib,*** and ***ic*** are the vertical stride for ***a, b,*** and ***c***. The following examples show a simple program that declares image data arrays and uses sub-arrays.

```
#include <alfast-libs.h>
#define NROWS 512
#define NCOLS 512

float a[NROWS*NCOLS];
float b[NROWS*NCOLS];
float c[NROWS*NCOLS];

/*--- sample1 - add image a to be giving c ---*/

void sample1 void
{
       Add (a, NCOLS, b, NCOLS, c, NCOLS, NROWS, NCOLS);
}

/*--- sample2 - add upper-left 128x128 subimage of a to upper
128x128
       subimage of b to lower-right 128x128 subimage of c ---*/
void sample2 void
{
       Add (a, NCOLS, b, NCOLS, c+128*NCOLS+128, NCOLS, 128, 128);
}
```

2. Common Input and Output Images

Many FastSeries Library functions may be called with the output image data specified being the same as one of the input images. For example **Add** may be called with the form:

Add (a, NCOLS, b, NCOLS, a, NCOLS, NROWS, NCOLS);

Not all functions may be called in this manner. Functions that disallow common input and output image data:

Conv3x3, Conv5x5, Conv7x7

Kirsch, Prewitt, Roberts, Sobel

Median, Median3x3, Median5x5, Median7x7

Rotate, Rotate8, Rotate16

Xgradient, Ygradient

E. FastSeries Library Reference

This chapter provides detailed documentation on the functions in the FastSeries Library. Each function is listed on a separate page showing input arguments, output arguments, strides, and execution functionality.

AbsDiff

AbsDiff Absolute value of difference

C Usage

```
#include <fastlib.h>
void AbsDiff (float *a, int ia, float *b, int ib, float *c, int
ic,
int nr, int nc)
```

Description

The **AbsDiff** function computes the absolute value of the difference between two input images. The following C fragment describes the function:

```
void AbsDiff (float *a, int ia, float *b, int ib, float *c, int
ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[i*ic+j] = fabs (a[i*ia+j]-b[i*ib+j]);
}
```

AbsSum

AbsSum Sum of absolute values

C Usage

```
#include <fastlib.h>
void AbsSum (float *a, int ia, float *b, int ib, float *c, int ic,
int nr, int nc)
```

Description

The **AbsSum** function computes the sum of the absolute values between two input images. The following C fragment describes the function:

```
void AbsSum (float *a, int ia, float *b, int ib, float *c, int ic,
int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[i*ic+j] = fabs (a[i*ia+j]) + fabs
                                                          (b[i*ib+j]);
}
```

Add

Add Add images

C Usage

```
#include <fastlib.h>
void Add (float *a, int ia, float *b, int ib, float *c, int ic,
int nr, int nc)
```

Description

The **Add** function computes the arithmetic sum of two input images. The following C fragment describes the function:

```
void Add (float *a, int ia, float *b, int ib, float *c, int ic,
int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[i*ic+j] = a[i*ia+j]+b[i*ib+j];
}
```

AddScalar

AddScalar - Add scalar to image

C Usage

```
#include <fastlib.h>
void AddScalar (float *a, int ia, float *b, float *c, int ic, int
nr,
int nc)
```

Description

The **AddScalar** function adds to each element of image ***a*** the scalar given by argument ***b***, placing the output at image ***c***. The following C code fragment describes the function:

```
void AddScalar (float *a, int ia, float *b, float *c, int ic, int
nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[i*ic+j] = a[i*ia+j] + *b;
}
```

And8

And8 Logical and of 8-bit image

C Usage

```
#include <fastlib.h>
void And8 (unsigned char *a, int ia, unsigned char *b, int ib,
unsigned char *c, int ic, int nr, int nc);
```

Description

The **And8** function performs a logical “and” of each element of image ***a*** with the corresponding element in image ***b,*** placing the output at image ***c***. The following C code fragment describes the function:

```
void And8 (unsigned char *a, int ia, unsigned char *b, int ib,
unsigned char *c, int ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = a[ia*i+j] & b[ib*i+j];
}
```

And16

And16 Logical and of 16 bit image

C Usage

```
#include <fastlib.h>
void And16 (unsigned short *a, int ia, unsigned short *b, int ib,
unsigned short *c, int ic, int nr, int nc)
```

Description

The **And16** function performs a logical “and” of each element of image ***a*** with the corresponding element in image ***b,*** placing the output at image ***c***. The following C code fragment describes the function:

```
void And16 (unsigned short *a, int ia, unsigned short *b, int ib,
unsigned short *c, int ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = a[ia*i+j] & b[ib*i+j];
}
```

AndNot8

AndNot8 - Logical AND 8-bit image with complement

C Usage

```
#include <fastlib.h>
void AndNot8 (unsigned char *a, int ia, unsigned char *b, int ib,
unsigned char *c, int ic, int nr, int nc)
```

Description

The **AndNot8** function performs a logical “and” of the complement of each element of image ***a*** with the corresponding element in image ***b,*** placing the output at image ***c***. The following C code fragment describes the function:

```
#include <fastlib.h>
void AndNot8 (unsigned char *a, int ia, unsigned char *b, int ib,
unsigned char *c, int ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i++)
              for (j = 0; j < nc; j++)
                     c[ic*i+j] = ~a[ia*i+j] & b[ib*i+j];
}
```

AndNot16

AndNot16 Logical AND 16 bit image with complement

C Usage

```
#include <fastlib.h>
void AndNot16 (unsigned short *a, int ia, unsigned short *b, int
ib,
unsigned short *c, int ic, int nr, int nc);
```

Description

The **AndNot16** function performs a logical “and” of the complement of each element of image ***a*** with the corresponding element in image ***b,*** placing the output at image ***c***. The following C code fragment describes the function:

```
void AndNot16 (unsigned short *a, int ia, unsigned short *b, int
ib, unsigned short *c, int ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = ~a[ia*i+j] & b[ib*i+j];
}
```

Ave

Ave Average float image

C Usage

```
#include <fastlib.h>
void Ave (float *a, int ia, float *b, int ib, float *c, float *d,
int id, int nr, int nc);
```

Description

The **Ave** function performs a “weighted average” of the two input float images. Each element of image ***a*** is scaled by ***c** and added to the corresponding element in image ***b,*** which has been scaled by **1.0 - *c.** The resultant image is placed at image ***d***. The following C code fragment describes the function:

```
#include <fastlib.h>
void Ave(float *a, int ia, float *b, int ib, float *c, float *d,
           int id, int nr, int nc)
{
int i;
int j;
float wt1, wt2;

       wt1 = *c;     / weighting fraction for latest image */
       wt2 = 1.0 - wt1;
       for (i = 0; i < nr; i ++)
       {
              for (j = 0; j < nc; j ++)
              {
                     d[i*id+j] = a[i*ia+j] * wt1 + b[i*ib+j] * wt2;
              }
       }
}
```

Ave8

Ave8 Average 8-bit image

C Usage

```
#include <fastlib.h>
void Ave8 (unsigned char *a, int ia, unsigned char *b, int ib,
float *c, unsigned char *d,
int id, int nr, int nc);
```

Description

The **Ave8** function performs a “weighted average” of the two input byte images. Each element of image ***a*** is scaled by ***c** and added to the corresponding element in image ***b,*** which has been scaled by **1.0 - *c.** The resultant image is placed at image ***d***. The following C code fragment describes the function:

```
#include <fastlib.h>
void Ave8(unsigned char *a, int ia, unsigned char *b, int ib,
float *c, unsigned char *d,
            int id, int nr, int nc)
{
int i;
int j;
float wt1, wt2;

       wt1 = *c;       / weighting fraction for latest image */
       wt2 = 1.0 - wt1;
       for (i = 0; i < nr; i ++)
       {
              for (j = 0; j < nc; j ++)
              {
                     d[i*id+j] = (unsigned char)(a[i*ia+j] * wt1 +
                                                 b[i*ib+j] * wt2);
              }
       }
}
```

Ave16

Ave16 Average 16 bit image

C Usage

```
#include <fastlib.h>
void Ave16 (unsigned short *a, int ia, unsigned short *b, int ib,
float *c, unsigned short *d, int id, int nr, int nc);
```

Description

The **Ave16** function performs a “weighted average” of the two input 16 bit images. Each element of image ***a*** is scaled by ***c** and added to the corresponding element in image ***b,*** which has been scaled by **1.0 - *c.** The resultant image is placed at image ***d***. The following C code fragment describes the function:

```
void Ave16(unsigned short *a, int ia, unsigned short *b, int ib,
float *c, unsigned short *d,
            int id, int nr, int nc)
{
int i;
int j;
float wt1, wt2;

       wt1 = *c;     / weighting fraction for latest image */
       wt2 = 1.0 - wt1;
       for (i = 0; i < nr; i ++)
       {
              for (j = 0; j < nc; j ++)
              {
              d[i*id+j] = (unsigned short)(a[i*ia+j] * wt1 +
                                           b[i*ib+j] * wt2);
              }
       }
}
```

bldwts

Summary

bldwts	build FFT weights

C Usage

```
void bldwts (int n);
```

Arguments

n	maximum FFT length; a power of 2

Description

Function **bldwts** creates a weight vector table used by the FFT routines (**cfft**, **cfftb**, **rfft2d**, etc.) consisting of complex exponential values.. Storage for the weight vector is allocated and freed dynamically off the memory heap.

Function **bldwts** must be called before any of the FFT routines are used. **bldwts** may be called multiple times with different table (i.e., vector) sizes, although generally **bldwts** is only called once with the largest size required by the application.

For multidimensional transforms, argument **n** to **bldwts** is determined by the largest of the array dimensions (in terms of complex elements) for the FFT. In the 1-D case, the argument **n** should be the largest vector length, in terms of complex elements, used by the application. Argument **n** must be a power of 2.

Example

```
#include <fastlib.h>
int n = 1024;

bldwts(n);
```

blkman

{xe”bldwts”}{xe “blkman”}

Summary

blkman	blackman window multiply

C Usage

```
void blkman (float *a, int ia, float *c, int ic, int n);
```

Arguments

A	Pointer to real vector a
ia	stride for vector a
c	Pointer to results in real vector c
ic	stride for vector c
n	element count of vectors

Description

blkman multiples real vector *a* by a Blackman window to condition it for performing an FFT according to the algorithm

```
for i=0 to n-1
      c[i] = a[i] * (0.42 - 0.5*cos(i*2*pi/n) +
0.08*cos(i*4*pi/n))
```

For other functions used for conditioning signals prior to performing an FFT, see the functions **hamm** and **hann**.

cdotpr

{xe”cdotpr”}{xe “blkman”}

Summary

cdotpr	complex dot product

C Usage

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

Arguments

a	pointer to complex vector a
ia	stride for vector a expressed in floats
b	pointer to complex vector b
ib	stride for vector b expressed in floats
c	pointer to result in complex scalar c
n	element count of vectors

Description

cdotpr computes the complex dot product of complex vectors *a* and *b* and stores the results in complex scalar *c* using the algorithm:

```
real(c) = sum(real(a[i]) * real(b[i]) - imag(a[i]) * imag(b[i]))
imag(c) = sum(real(a[i]) * imag(b[i]) + imag(a[i]) * real(b[i]))
```

Note that the stride arguments *ia* and *ib* are expressed in floats. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

cfft

{xe”cfft”}{xe “blkman”}

Summary

cfft	complex FFT (in place)

C Usage

```
void cfft (float *a, int n, int direction);
```

Arguments

a	pointer to complex vectors a containing real and imaginary elements. The results are stored in *a*.
n	element count of vector must be a power of 2
direction	processing mode direction = 1 for forward FFT direction = -1 for inverse FFT

Description

cfft computes either a forward or inverse complex FFT on the data in complex vector *a* and stores the results back into vector *a*. *direction* determines the form of the FFT.

If *direction* = 1, the function performs a forward FFT. The results are not scaled. (They may be scaled using **cfftsc** to multiply by *1/n*.) If *direction* = -1, the function performs an inverse FFT. In this case, the results do not need to be scaled.

A call must have been previously made to the function **bldwts** in order to build a weight vector required by the FFT functions. **bldwts** need only be called once, with an argument specifying the maximum FFT vector length required in calls to the **cfft** routine. In other words, **bldwts** should be set up with the greatest value of argument **n** that will be used during the application.

cfft2d

{xe”cfft2d”}{xe “blkman”}

Summary

cfft2d	complex 2D FFT (in place)

C Usage

```
void cfft2d (float *a, int nr, int nc, int direction);
```

Arguments

a	input/output complex matrix a
nr	number of rows in a; power of 2
nc	number of columns in a; power of 2
direction	processing mode direction = 1 for forward FFT direction = -1 for inverse FFT

Description

cfft2d computes either a forward or inverse complex two dimensional FFT on the data in complex matrix *a* and stores the results back into matrix *a*.

Argument *direction* determines the form of the FFT. If *direction* = 1, the function performs a forward FFT. The results are not scaled, and they may be scaled using **cfftsc** to multiply by *1/(nr*nc)*. If *direction* = -1, the function performs an inverse FFT. The results do not need to be scaled.

A call must have been previously made to the function **bldwts** in order to build a weight vector required by the FFT functions. **bldwts** need only be called once, with an argument specifying the maximum FFT length required in calls to the **cfft2d** routine. This would be the greater of **nr** and **nc**.

Note that the FFT functions perform on row-major data. Users with data in column-major order may reverse the row and column arguments to produce the desired effect.

cfftb

{xe"cfftb"}{xe "blkman"}

Summary

cfftb	complex FFT (not in place)

C Usage

```
void cfftb (float *a, float *c, int n, int direction);
```

Arguments

A	pointer to complex vector a.
C	pointer to results in complex vector c.
N	element count of vector. must be a power of 2
direction	processing mode direction = 1 for forward FFT direction = -1 for inverse FFT

Description

cfftb computes either a forward or inverse complex FFT on the data in complex vector *a* and stores the results into complex vector *c*. *direction* determines the form of the FFT.

If *direction* = 1, the function performs a forward FFT. The results are not scaled, and they may be scaled using **cfftsc** to multiply by *1/n*. If *direction* = -1, the function performs an inverse FFT. The results do not need to be scaled.

A call must have been previously made to the function **bldwts** in order to build a weight vector required by the FFT functions. **bldwts** need only be called once, with an argument specifying the maximum FFT length required in calls to the **cfftb** routine. In other words, **bldwts** should be called with no less than the maximum value for argument **n** that will be used by the application.

cfftsc

{xe”cfftsc”}{xe “blkman”}

Summary

cfftsc	complex FFT scale

C Usage

```
void cfftsc (float *c, int n);
```

Arguments

c	complex vector c. results stored in vector c
n	element count of vector

Description

cfftsc scales the FFT in complex vector *c* by dividing the real and imaginary part of each element by *n* and storing the results back into vector *c*.

cfftss

{xe”cfftss”}{xe “blkman”}

Summary

cfftss	complex FFT scale with stride

C Usage

```
void cfftss (float *c, int ic, int n);
```

Arguments

c	pointer to complex vector c. results stored in vector c
ic	stride for vector c expressed in floats
n	element count of vector

Description

cfftss uses the stride value to scale elements of the FFT in complex vector *c* by dividing the real and imaginary part of the elements by *n* and storing the results back into vector *c*. Note that the *stride* argument is expressed in floats. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

cimul

{xe”cimul”}{xe “blkman”}

Summary

cimul	complex image multiplication

C Usage

```
void cimul (float *a, float *b, float *c, int nrows, int ncols,
int direction);
```

Arguments

a	pointer to complex array a
b	pointer to complex array b
c	pointer to results in complex array c
nrows	row count of arrays a, b, and c
ncols	column count of arrays a, b, and c
direction	processing mode direction = 1 for normal processing direction= -1 for conjugate processing

Description

cimul multiplies two complex arrays *a* and *b* and stores the results in complex array *c* using the algorithm

```
for i=0 to nrows-1, j=0 to ncols-1
       if direction == 1
              real(c[i,j])   = real(a[i,j])*real(b[i,j]) –
                                    imag(a[i,j]) * imag(b[i,j])
              imag(c[i,j])   = real(a[i,j])*imag(b[i,j]) +
                                    imag(a[i,j]) * real(b[i,j])
       else
              real(c[i,j])   = real(a[i,j])*real(b[i,j]) +
                                    imag(a[i,j]) * imag(b[i,j])
              imag(c[i,j])   = -real(a[i,j])*imag(b[i,j]) +
                                    imag(a[i,j]) * real(b[i,j])
```

conv

{xe”conv”}{xe “blkman”}

Summary

Conv	convolution and correlation

C Usage

```
void conv (float *a, int ia, float *b, int ib, float *c, int ic,
int nc, int nb, int ndec);
```

Arguments

a	pointer to real vector a of. length (nc-1)*ndec+nb
ia	stride for vector a
b	pointer to real vector b
ib	stride for vector b ib > 0 to perform correlation ib < 0 to perform convolution
c	pointer to results in real vector c
ic	stride for vector c
nc	element count of vector c
nb	element count of vector b
ndec	decimation factor

Description

conv either performs the convolution or correlation of real vectors *a* and *b* and stores the results in real vector *c*. If *ib* is positive, the function performs the correlation. If *ib* is negative, the function performs a convolution. Note that *b* must point to the last element in the vector in that case. The *ndec* decimation factor specifies what portion of the output value is actually computed. A value of *ndec* =3 indicates that only every third possible output value is computed.

The algorithm for correlation is

```
for i=0 to nc-1, j=0 to nb-1
       c[i] = sum( a[ndec*i + j*ia] * b[j*ib] )
```

The algorithm for convolution is

```
for i=0 to nc-1, j=0 to nb-1
       c[i] = sum( a[ndec*i + j*ia] * b[nb-(j-1)*ib] )
```

Conv2D

Conv2D - General 2D convolution

C Usage

```
#include <fastlib.h>
void Conv2D (float *a, int ia, float *knl, int kr, int kc,
float *c, int ic, int nr, int nc);
```

Description

The **Conv2D** function performs a two-dimensional convolution of the input kernel, **knl**, with the input image **a**. The output image is placed at **c.**

Conv3x3

Conv3x3 3x3 convolution

C Usage

```
include <fastlib.h>
void Conv3x3 (float *a, int ia, float *c, int ic, int nr, int nc,
float *knl);
```

Description

The **Conv3x3** function performs a 3 by 3 convolution of image ***a*** with the 3x3 kernel located in ***knl,*** placing the output at image ***c***.

Conv5x5

Conv5x5 5x5 convolution

C Usage

```
#include <fastlib.h>
void Conv5x5 (float *a, int ia, float *c, int ic, int nrow, int
ncol,
float *knl);
```

Description

The **Conv5x5** function performs a 5 by 5 convolution of image ***a*** with the 5x5 kernel located in ***knl,*** placing the output at image ***c***.

Conv7x7

Conv7x7 7x7 convolution

C Usage

```
#include <fastlib.h>
void Conv7x7 (float *a, int ia, float *c, int ic, int nrow, int
ncol,
float *knl);
```

Description

The **Conv7x7** function performs a 7 by 7 convolution of image ***a*** with the 7x7 kernel located in ***knl,*** placing the output at image ***c***.

ConvFFT

ConvFFT Convolution using FFT on float image

C Usage

```
#include <fastlib.h>
void ConvFFT(float *a, int ia, float *knl, int kr, int kc,
float *c, int ic, int nr, int nc);
```

Description

The **ConvFFT** performs a circular convolution of image **a** with a kernel **(knl)** by the multiplication of the two functions in the frequency domain. The output image is placed at **c**.

cvadd

{xe”cvadd”}{xe “blkman”}

Summary

cvadd	complex vector add

C Usage

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

Arguments

A	pointer to complex vector a
Ia	stride for vector a expressed in floats
b	pointer to complex vector b
ib	stride for vector b expressed in floats
c	pointer to results in complex vector c
ic	stride for vector c expressed in floats
n	element count of vectors

Description

cvadd adds complex vectors *a* and *b* and stores the results in complex vector *c* using the algorithm below.

```
for i=0 to n-1
       real(c[i]) = real(a[i]) + real(b[i])
       imag(c[i]) = imag(a[i]) + imag(b[i])
```

Note that the *stride* argument(s) to complex vectors are expressed in floats. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

cvcomb

{xe”cvcomb”}{xe “blkman”}

Summary

Cvcomb	complex vector combine

C Usage

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

Arguments

a	pointer to real vector a
ia	stride for vector a
b	pointer to real vector b
ib	stride for vector b
c	pointer to results in complex vector c
ic	stride for vector c expressed in floats
n	element count of vectors

Description

cvcomb uses the elements of real vectors *a* and *b* to form complex vector *c*. The real elements of *c* are taken from vector *a*; the imaginary elements are taken from vector *b.* Note that the *stride* argument(s) to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

```
for i=0 to n-1
       real (c[i]) = a[i]
       imag (c[i]) = b[i]
```

cvconj

{xe”cvconj”}{xe “blkman”}

Summary

cvconj	complex vector conjugate

C Usage

```
void cvconj (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	pointer to complex vector a
ia	stride for vector a expressed in floats
c	pointer to results in complex vector c
ic	stride for vector c expressed in floats
n	element count of vectors

Description

cvconj conjugates the element of complex vector *a* and stores the results in complex vector *c.* Note that the *stride* argument(s) to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

```
for i=0 to n-1
       real(c[i]) = real(a[i])
       imag(c[i]) = -imag(a[i])
```

cvcsml

{xe”cvcsml”}{xe “blkman”}

Summary

cvcsml	complex vector complex scalar multiply

C Usage

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

Arguments

a	pointer to complex vector a
ia	stride for vector a expressed in floats
b	pointer to complex scalar b
c	pointer to results in complex vector c
ic	stride for vector c expressed in floats
n	element count of vectors

Description

cvcsml multiplies elements of complex vector *a* by complex scalar *b* and stores the result into complex vector *c* using the algorithm below. Note that the *stride* arguments to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

```
for i=0 to n-1
       real(c[i]) = real(a[i])*real(b) - imag(a[i])*imag(b)
       imag(c[i]) = real(a[i])*imag(b) + imag(a[i])*real(b)
```

cvdiv

{xe”cvdiv”}{xe “blkman”}

Summary

cvdiv	complex vector divide

C Usage

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

Arguments

a	pointer to complex vector a
ia	stride for vector a expressed in floats
b	pointer to complex vector b
ib	stride for vector b expressed in floats
c	pointer to results in complex vector c
ic	stride for vector c expressed in floats
n	element count of vectors

Description

cvdiv divides complex vector *a* by complex vector *b* and stores the results in complex vector *c*. The vector b may not contain elements with both real and complex values of 0.0. Note that the *stride* argument(s) to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4. The division algorithm is

```
for i=0 to n-1
       real(c[i]) = (real(a[i]) * real(b[i]) + imag(a[i]) *
              .imag(b[i])) /(real(b[i])**2 + imag(b[i])**2)
       imag(c[i]) = (imag(a[i]) * real(b[i]) - real(a[i]) *
               imag(b[i])) /(real(b[i])**2 + imag(b[i])**2)
```

cvexp

{xe”cvexp”}{xe “blkman”}

Summary

cvexp	complex vector exponential

C Usage

```
void cvexp (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	pointer to real vector a
ia	Stride for vector a
c	pointer to results in complex vector c
ic	Stride for vector c expressed in floats
n	element count of vectors

Description

cvexp computes the complex exponential of real vector *a* and stores the results in complex vector *c* according to the algorithm:

```
for i=0 to n-1
       real(c[i]) = cos(a[i])
       imag(c[i]) = sin(a[i])
```

Note that the *stride* argument(s) to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

cvmags

{xe”cvmags”}{xe “blkman”}

Summary

cvmags	complex vector magnitude squared

C Usage

```
void cvmags (float *a, int ia, float *c, int ic, int n);
```

Arguments

A	Pointer to complex vector a
ia	stride for vector a expressed in floats
C	Pointer to results in real vector c
ic	stride for vector c
N	element count of vectors

Description

cvmags computes square of the magnitude of each element of complex vector *a* and stores the results in real vector *c*. The square of the magnitude is

```
for i=0 to n-1
       c[i] = real(a[i])**2 + imag(a[i])**2
```

Note that the *stride* argument(s) to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

cvmov

{xe”cvmov”}{xe “blkman”}

Summary

cvmov	complex vector move

C Usage

```
void cvmov (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	pointer to complex vector a
ia	stride for vector a expressed in reals
c	pointer to results in complex vector c
ic	stride for vector c expressed in reals
n	element count of vectors

Description

cvmov moves the elements of complex vector *a* to complex vector *c*

```
for i=0 to n-1
       real(c[i]) = real(a[i])
       imag(c[i]) = imag(a[i])
```

Note that the *stride* argument(s) to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

cvmul

{xe”cvmul”}{xe “blkman”}

Summary

cvmul	complex vector multiply

C Usage

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

Arguments

a	pointer to complex vector a
ia	stride for vector a expressed in floats
b	pointer to complex vector b
ib	stride for vector b expressed in floats
c	pointer to results in complex vector c
ic	stride for vector c expressed in floats
n	element count of vectors expressed in complex elements
direction	processing mode direction = 1 for normal multiply direction = -1 for multiply with conjugate

Description

For *direction* = 1, **cvmul** does a normal complex multiply of each element of two complex vectors *a* and *b* and stores the results in complex vector *c* using the algorithm

```
for i=0 to n-1
       real(c[i]) = real(a[i]) * real(b[i]) - imag(a[i]) *
                                                 imag(b[i])
       imag(c[i]) = real(a[i]) * imag(b[i]) + imag(a[i]) *
                                                 real(b[i])
```

For *direction* = -1, **cvmul** multiplies vector *b* by the conjugate of vector *a* with the algorithm

```
for i=0 to n-1
       real(c[i]) = real(a[i]) * real(b[i]) + imag(a[i]) *
                                                 imag(b[i])
       imag(c[i]) = real(a[i]) * imag(b[i]) - imag(a[i]) *
                                                 real(b[i])
```

Note that the *stride* argument(s) to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

cvneg

{xe”cvneg”}{xe “blkman”}

Summary

cvneg	complex vector negate

C Usage

```
void cvneg (float *a, int ia, float *c, int ic, int n);
```

Arguments

A	pointer to complex vector a
Ia	stride for vector a expressed in floats
C	pointer to results in complex vector c
Ic	stride for vector c expressed in floats
N	element count of vectors

Description

cvneg negates the elements of complex vector *a* and stores the results in complex vector *c*

```
for i=0 to n-1
       real(c[i]) = -real(a[i])
       imag(c[i]) = -imag(a[i])
```

Note that the *stride* argument(s) to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

cvrcip

{xe”cvrcip”}{xe “blkman”}

Summary

cvrcip	complex vector reciprocal

C Usage

```
void cvrcip (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	pointer to complex vector a
ia	stride for vector a expressed in floats
c	pointer to results in complex vector c
ic	stride for vector c expressed in floats
n	element count of vectors

Description

cvrcip computes the reciprocal of the elements of complex vector *a* and stores the results in complex vector *c*

```
for i=0 to n-1
       real(c[i]) = real(a[i]) / (real(a[i])**2 + imag(a[i])**2)
       imag(c[i]) = -imag(a[i]) / (real(a[i])**2 + imag(a[i])**2)
```

Note that the *stride* argument(s) to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

cvsma

{xe”cvsma”}{xe “blkman”}

Summary

cvsma	complex vector scalar multiply and add

C Usage

```
void cvsma (float *a, int ia, float *b, float *c, int ic,
float *d, int id, int n);
```

Arguments

A	pointer to complex vector a
Ia	stride for vector a expressed in floats
B	pointer to complex scalar b
C	pointer to complex vector c
Ic	stride for vector c expressed in floats
D	pointer to results in complex vector d
id	stride for vector d expressed in floats
N	element count of vectors

Description

cvsma multiplies elements of complex vector *a* by complex scalar *b*, adds the corresponding element of complex vector *c*, and stores the result into complex vector *d* using the algorithm

```
for i=0 to n-1
       real(d[i]) = real(a[i])*real(b) - imag(a[i])*imag(b) +
                                                 real(c[i])
       imag(d[i]) = real(a[i])*imag(b) + imag(a[i])*real(b) +
                                                 imag(c[i])
```

Note that the *stride* argument(s) to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

cvsub

{xe”cvsub”}{xe “blkman”}

Summary

cvsub	complex vector subtract

C Usage

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

Arguments

A	pointer to complex vector a
ia	stride for vector a expressed in floats
B	pointer to complex vector b
ib	stride for vector b expressed in floats
C	pointer to results in complex vector c
ic	stride for vector c expressed in floats
N	element count of vectors

Description

cvsub subtracts complex vector *b* from complex vector *a* and stores the results in complex vector *c*

```
for i=0 to n-1
       real(c[i]) = real(a[i]) - real(b[i])
       imag(c[i]) = imag(a[i]) - imag(b[i])
```

Note that the *stride* argument(s) to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

Div

Div Divide images

C Usage

```
#include <fastlib.h>
void Div (float *a, int ia, float *b, int ib, float *c, int ic,
                                          int nr, int nc);
```

Description

The **Div** function will divide each pixel of image ***a*** with the corresponding pixel located in ***b,*** placing the output at image ***c***. The following C code fragment describes the function:

```
#include <fastlib.h>
void Div (float *a, int ia, float *b, int ib, float *c, int ic,
int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[i*ic+j] = a[i*ia+j]/b[i*ib+j];
}
```

dotpr

{xe”doptpr”}{xe “blkman”}

Summary

Dotpr	dot product

C Usage

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

Arguments

a	pointer to real vector a
ia	stride for vector a
b	pointer to real vector b
ib	stride for vector b
c	pointer to result in real scalar c
n	element count of a and b

Description

dotpr computes the dot product of two real vectors *a* and *b* and stores the result in real scalar *c* using the algorithm

```
ainc = 0
binc = 0
for i=0 to n-1
       c = sum (a[ainc] * b[binc])
       ainc = ainc + ai
       binc = binc + bi
```

fastlibversion

{xe”fastlibversion”}{xe “blkman”}

Summary

fastlibversion	FastSeries Library version

C Usage

```
extern char fastlibversion[ ];
```

Description

vlibversion is a character array containing a string specifying the FastSeries Library version.

The following C example demonstrates the use of **fastlibversion**.

```
void prt_version void
{
       extern char fastlibversion[ ];

       printf (“%s\n”, fastlibversion);
}
```

Fill

Fill Fill float image

C Usage

```
#include <fastlib.h>
void Fill (float *a, int ia, float *b, int nr, int nc);
```

Description

The **Fill** function loads each pixel of the float image ***a*** with the value located at ***b***. The following C code fragment describes the function:

```
void Fill (float *a, int ia, float *b, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     a[ia*i+j] = *b;
}
```

Fill8

Fill8 Fill 8-bit image

C Usage

```
#include <fastlib.h>
void Fill8 (unsigned char *a, int ia, unsigned char *b, int nr,
int nc);
```

Description

The **Fill8** function will load each pixel of the 8-bit image ***a*** with the value located at ***b***. The following C code fragment describes the function:

```
void Fill8 (unsigned char *a, int ia, unsigned char *b, int nr,
int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     a[ia*i+j] = *b;
}
```

Fill16

Fill16 Fill 16 bit image

C Usage

```
#include <fastlib.h>
void Fill16 (unsigned short *a, int ia, unsigned short *b,
int nr, int nc);
```

Description

The **Fill16** function will load each pixel of the 16-bit image ***a*** with the value located at ***b***. The following C code fragment describes the function:

```
#include <fastlib.h>
void Fill16 (unsigned short *a, int ia, unsigned short *b, int nr,
int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     a[ia*i+j] = *b;
}
```

fixinta

{xe”fixinta”}{xe “blkman”}

Summary

fixinta	fix pixels in format A (high/low)

C Usage

```
void fixinta (float *a, int ia, unsigned short *c, int n);
```

Arguments

a	pointer to real vector a
ia	stride for vector a
c	pointer to results in pixel vector c
n	element count of vector

Description

fixinta converts the elements of real vector *a* to 8-bit packed integers in a high/low format. Each element is fixed and the low-order 8 bits of the integer are taken as a positive magnitude. They are then stored as pairs into short integers in unsigned short vector *c* in least-significant then most-significant order. The algorithm can be written as follows0

```
for i=0 to n-1 in steps of 2
       c[i/2] = fix(a[i]).AND.$FF .OR. 256*fix(a[i+1]).AND.$FF
```

fixintb

{xe”fixintb”}{xe “blkman”}

Summary

fixintb	fix pixels in format B (low/high)

C Usage

```
void fixintb (float *a, int ia, unsigned short *c, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	pointer to results in pixel vector c
n	element count of vector

Description

fixintb converts the elements of real vector *a* to 8-bit packed integers in a low/high format. Each element is fixed and the low-order 8 bits of the integer are taken as a positive magnitude. They are then stored as pairs into short integers in short vector *c* in most-significant then least-significant order. The algorithm can be written as follows

```
for i=0 to n-1 in steps of 2
       c[i/2] = 256*fix(a[i]).AND.$FF .OR. fix(a[i+1]).AND.$FF
```

Fix8

Fix8 Convert float image to 8-bit integer

C Usage

```
#include <fastlib.h>
void Fix8 (float *a, int ia, unsigned char *c, int ic, int nr, int
nc);
```

Description

The **Fix8** function will convert a float image stored at ***a*** to an 8-bit image, which will be placed at ***b***. The following C code fragment describes the function:

```
void Fix8 (float *a, int ia, unsigned char *c, int ic, int nr, int
nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = (unsigned char) a[ia*i+j];
}
```

Fix16

Fix16 Convert float image to 16 bit integer

C Usage

```
#include <fastlib.h>
void Fix16 (float *a, int ia, unsigned short *c, int ic, int nr,
int nc);
```

Description

The **Fix16** function will convert a float image stored at ***a*** to an 16-bit image, which will be placed at ***b***. The following C code fragment describes the function:

```
void Fix16 (float *a, int ia, unsigned short *c, int ic, int nr,
int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = (unsigned short) a[ia*i+j];
}
```

Float8

Float8 Convert 8-bit integer image to float

C Usage

```
#include <fastlib.h>
void Float8 (unsigned char *a, int ia, float *c, int ic, int nr,
int nc);
```

Description

The **Float8** function will convert an 8-bit image stored at ***a*** to a float image, which will be placed at ***b***. The following C code fragment describes the function:

```
void Float8 (unsigned char *a, int ia, float *c, int ic, int nr,
int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = a[ia*i+j];
}
```

Float16

Float16 Convert 16 bit integer image to float

C Usage

```
#include <fastlib.h>
void Float16 (unsigned short *a, int ia, float *c, int ic, int nr,
int nc);
```

Description

The **Float16** function will convert a 16-bit image stored at ***a*** to a float image, which will be placed at ***b***. The following C code fragment describes the function:

```
void Float16 (unsigned short *a, int ia, float *c, int ic, int nr,
int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = a[ia*i+j];
}
```

fltinta

{xe”fitinta”}{xe “blkman”}

Summary

fltinta	float pixels in format A (high/low)

C Usage

```
void fltinta (unsigned short *a, float *c, int ic, int n);
```

Arguments

A	pixel vector a
C	results in real vector c
ic	stride for vector c
n	element count of vector

Description

fltinta converts the elements of pixel vector *a* from 8-bit packed integers in a high/low format to real values and stores them into vector *c*. Each 8-bit element in vector *a* is taken as a positive magnitude and floated. They are extracted in least-significant then most-significant order. The algorithm can be written as follows

```
for i=0 to n-1 in steps of 2
       c[i] = float(a[i/2]).AND.$FF)
       c[i+1] = float(a[i/2]).AND.$FF00)/256
```

fltintb

{xe”fttintb”}{xe “blkman”}

Summary

fltintb	float pixels in format B (low/high)

C Usage

```
void fltintb (unsigned short *a, float *c, int ic, int n);
```

Arguments

a	pixel vector a
c	results in real vector c
ic	stride for vector c
n	element count of vector

Description

fltintb converts the elements of pixel vector *a* from 8-bit packed integers in a low/high format to real values and stores them into vector *c*. Each 8-bit element in vector *a* is taken as a positive magnitude and floated. They are extracted in most-significant then least-significant order. The algorithm can be written as follows

```
for i=0 to n-1 in steps of 2
       c[i] = float(a[i/2]).AND.$FF00)/256
       c[i+1] = float(a[i/2]).AND.$FF)
```

hamm

{xe”hamm”}{xe “blkman”}

Summary

hamm	hamming window multiply

C Usage

```
void hamm (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

hamm multiplies real vector *a* by a Hamming window and stores the resulting real vector into *c*. The Hamming algorithm is

```
for i-0 to n-1
       c[i] = a[i] * (0.54 - 0.46*cos(i*2*pi/n))
```

blkman and **hann** are two other functions for conditioning signals prior to performing an FFT.

hann

{xe”hann”}{xe “blkman”}

Summary

hann	hanning window multiply

C Usage

```
void hann (float *a, int ia, float *c, int ic, int n, int flag);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors
flag	normalization mode flag = 0 for unnormalized hanning window flag = 1 for normalized hanning window

Description

For *flag* = 0, **hann** multiplies real vector *a* by an unnormalized Hanning window using the algorithm

```
for i=0 to n-1
       c[i] = 0.5 * a[i] * (1.0 - cos(i*2*pi/n))
```

For *flag* = 1, **hann** multiplies real vector *a* by a normalized Hanning window by the algorithm

```
for i=0 to n-1
       c[i] = sqrt(2.0/3.0) * a[i] * (1.0 -
                    cos(i*2*pi/n))
```

blkman and **hamm** are two other functions used for conditioning signals prior to performing an FFT.

hist

{xe”hiht”}{xe “blkman”}

Summary

hist histogram

C Usage

```
void hist (float *a, int ia, float *c, int n, int nc,
float *amax, float *amin);
```

Arguments

a	real vector a
ia	stride for vector a
c	histogram in real vector c
n	element count of vector a
nc	number of bins in c
amax	maximum histogram value
amin	minimum histogram value

Description

hist constructs a histogram on the elements of real vector *a* and adds the results to the histogram in vector *c*. The number of bins in the histogram and the maximum and minimum values of the range of interest is specified by *nc*, *amax*, and *amin*, respectively. The width of each bin is (*amax* - *amin*)/ *nc* except that values below *amin* or above *amax* are counted in the first and last bin, respectively.

Note that upon entry to hist, vector *c* contains an initial histogram. Upon return, vector *c* has been updated with the number of elements of vector *a* that fell in each bin according to the algorithm

```
for i=0 to n-1
       if a[i] < amin,  j = 0,
       else if a[i] >= amax,  j = nc-1,
       else,  j = int(nc*(a[i]-amin)/(amax-amin)),
       then  c[j] = c[j] + 1.0
```

HistoEqual8

HistoEqual8 Histogram Equalization of 8-bit image

C Usage

```
#include <fastlib.h>
void HistoEqual8 (unsigned char *a, int ia, u_long *b, unsigned
                                              char *c, int ic,
                  int nr, int nc);
```

Description

The **HistoEqual8** function will perform a histogram equalization on the 8-bit image located at ***a***. The function must be passed the histogram for image ***a.*** The user must place the histogram at ***b***. The output image is placed at location ***c***. The following C code fragment describes the function:

```
#define MAX8BIT     255
void HistoEqual8 (unsigned char *a, int ia, u_long *b, unsigned
char *c, int ic, int nr, int nc)
{
int i;
int j;
u_long pixel_count, total;
unsigned char tran_fn[MAX8BIT + 1];

/* find the total pixel count */
pixel_count = nr * nc;

/* create image transform lut*/
for (i = 0, total = 0; i <= MAX8BIT; i++)
       {
              total += b[i];  /* calc. cumulative histogram */
              tran_fn[i] = (unsigned char)(MAX8BIT *
                  (float)((float)total/(float)pixel_count) + 0.5);
       }

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = tran_fn[a[ia*i+j]];
}
```

HistoEqual10

HistoEqual10 Histogram Equalization of 10 bit image

C Usage

```
#include <fastlib.h>
void HistoEqual10 (unsigned short *a, int ia, u_long *b,
                   unsigned short *c, int ic, int nr, int nc);
```

Description

The **HistoEqual10** function will perform a histogram equalization on the 10-bit image located at ***a***. The function must be passed the histogram for image ***a.*** The user must place the histogram at ***b***. The output image is placed at location ***c***. The following C code fragment describes the function:

```
#define MAX10BIT    1023
void HistoEqual10 (unsigned short *a, int ia, u_long *b, unsigned
short *c, int ic, int nr, int nc)
{
int i;
int j;
u_long pixel_count, total;
unsigned short tran_fn[MAX10BIT + 1];

/* find the total pixel count */
pixel_count = nr * nc;

/* create image transform lut*/
for (i = 0, total = 0; i <= MAX10BIT; i++)
      {
            total += b[i];  /* calc. cumulative histogram */
            tran_fn[i] = (unsigned char)(MAX10BIT *
               (float)((float)total/(float)pixel_count) + 0.5);
      }

      for (i = 0; i < nr; i ++)
            for (j = 0; j < nc; j ++)
                  c[ic*i+j] = tran_fn[a[ia*i+j]];
}
```

HistoEqual12

HistoEqual12 Histogram Equalization of 12 bit image

C Usage

```
#include <fastlib.h>
void HistoEqual12 (unsigned short *a, int ia, u_long *b,
                   unsigned short *c, int ic, int nr, int nc);
```

Description

The **HistoEqual12** function will perform a histogram equalization on the 12-bit image located at ***a***. The function must be passed the histogram for image ***a.*** The user must place the histogram at ***b***. The output image is placed at location ***c***. The following C code fragment describes the function:

```
#define MAX12BIT    4095
void HistoEqual12 (unsigned short *a, int ia, u_long *b, unsigned
short *c, int ic, int nr, int nc)
{
int i;
int j;
u_long pixel_count, total;
unsigned short tran_fn[MAX12BIT + 1];

/* find the total pixel count */
pixel_count = nr * nc;

/* create image transform lut*/
for (i = 0, total = 0; i <= MAX12BIT; i++)
       {
              total += b[i];  /* calc. cumulative histogram */
              tran_fn[i] = (unsigned char)(MAX12BIT *
                 (float)((float)total/(float)pixel_count) + 0.5);
       }

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = tran_fn[a[ia*i+j]];
}
```

HistoEqual14

HistoEqual14 Histogram Equalization of 14 bit image

C Usage

```
#include <fastlib.h>
void HistoEqual14 (unsigned short *a, int ia, u_long *b,
unsigned short *c, int ic, int nr, int nc);
```

Description

The **HistoEqual14** function will perform a histogram equalization on the 14-bit image located at ***a***. The function must be passed the histogram for image ***a.*** The user must place the histogram at ***b***. The output image is placed at location ***c***. The following C code fragment describes the function:

```
#define MAX14BIT    16383
void HistoEqual14 (unsigned short *a, int ia, u_long *b, unsigned
short *c, int ic, int nr, int nc)
{
int i;
int j;
u_long pixel_count, total;
unsigned short tran_fn[MAX14BIT + 1];

/* find the total pixel count */
pixel_count = nr * nc;

/* create image transform lut*/
for (i = 0, total = 0; i <= MAX14BIT; i++)
      {
            total += b[i];  /* calc. cumulative histogram */
            tran_fn[i] = (unsigned char)(MAX14BIT *
               (float)((float)total/(float)pixel_count) + 0.5);
      }

      for (i = 0; i < nr; i ++)
      for (j = 0; j < nc; j ++)
            c[ic*i+j] = tran_fn[a[ia*i+j]];
}
```

HistoEqual16

HistoEqual16 Histogram Equalization of 16 bit image

C Usage

```
#include <fastlib.h>
void HistoEqual16 (unsigned short *a, int ia, u_long *b,
unsigned short *c, int ic, int nr, int nc)
```

Description

The **HistoEqual16** function will perform a histogram equalization on the 16-bit image located at ***a***. The function must be passed the histogram for image ***a.*** The user must place the histogram at ***b***. The output image is placed at location ***c***. The following C code fragment describes the function:

```
#include <fastlib.h>
#define MAX16BIT    65535

void HistoEqual16 (unsigned short *a, int ia, u_long *b, unsigned
short *c, int ic, int nr, int nc)
{
int i;
int j;
u_long pixel_count, total;
unsigned short tran_fn[MAX16BIT + 1];

/* find the total pixel count */
pixel_count = nr * nc;

/* create image transform lut*/
for (i = 0, total = 0; i <= MAX16BIT; i++)
       {
       total += b[i];  /* calc. cumulative histogram */
              tran_fn[i] = (unsigned char)(MAX16BIT *
                  (float)((float)total/(float)pixel_count) + 0.5);
       }

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = tran_fn[a[ia*i+j]];
}
```

Histogram8

Histogram8 Calculate histogram for 8-bit image

C Usage

```
#include <fastlib.h>
void Histogram8 (unsigned char *a, int ia, u_long *c, int nr, int
nc);
```

Description

The **Histogram8** function will calculate the histogram for the 8-bit image located at ***a***. The histogram will be placed at ***c.*** The following C code fragment describes the function:

```
#include <fastlib.h>
void Histogram8 (unsigned char *a, int ia, u_long *c, int nr, int
nc)
{
int i;
int j;

       for (i = 0; i < 256; i ++)
              c[i] = 0;
       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[a[ia*i+j]] ++;
}
```

Histogram10

Histogram10 Calculate histogram for 10 bit image

C Usage

```
#include <fastlib.h>
void Histogram10 (unsigned short *a, int ia, u_long *c, int nr,
int nc);
```

Description

The **Histogram10** function will calculate the histogram for the 10-bit image located at ***a***. The histogram will be placed at ***c.*** The following C code fragment describes the function:

```
void Histogram10 (unsigned short *a, int ia, u_long *c, int nr,
int nc)
{
int i;
int j;

       for (i = 0; i < 1024; i ++)
              c[i] = 0;
       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[a[ia*i+j]] ++;

}
```

Histogram12

Histogram12 Calculate histogram for 12 bit image

C Usage

```
#include <fastlib.h>
void Histogram12 (unsigned short *a, int ia, u_long *c, int nr,
int nc);
```

Description

The **Histogram12** function will calculate the histogram for the 12-bit image located at ***a***. The histogram will be placed at ***c.*** The following C code fragment describes the function:

```
#include <fastlib.h>
void Histogram12 (unsigned short *a, int ia, u_long *c, int nr,
int nc)
{
int i;
int j;

       for (i = 0; i < 4096; i ++)
              c[i] = 0;
       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[a[ia*i+j]] ++;

}
```

Histogram14

Histogram14 Calculate histogram for 14 bit image

C Usage

```
#include <fastlib.h>
void Histogram14 (unsigned short *a, int ia, u_long *c, int nr,
int nc);
```

Description

The **Histogram14** function will calculate the histogram for the 14-bit image located at ***a***. The histogram will be placed at ***c.*** The following C code fragment describes the function:

```
void Histogram14 (unsigned short *a, int ia, u_long *c, int nr,
int nc)
{
int i;
int j;

       for (i = 0; i < 16384; i ++)
              c[i] = 0;
       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[a[ia*i+j]] ++;

}
```

Histogram16

Histogram16 Calculate histogram for 16 bit image

C Usage

```
#include <fastlib.h>
void Histogram16 (unsigned short *a, int ia, u_long *c, int nr,
int nc);
```

Description

The **Histogram16** function will calculate the histogram for the 16-bit image located at ***a***. The histogram will be placed at ***c.*** Histogram16 may require the stack size to be increased. RT860 -s option can be used to set the stack size. The following C code fragment describes the function:

```
void Histogram16 (unsigned short *a, int ia, u_long *c, int nr,
int nc)
{
int i;
int j;

       for (i = 0; i < 65536; i ++)
              c[i] = 0;
       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[a[ia*i+j]] ++;

}
```

Histogram

Histogram Calculate histogram of float image

C Usage

```
#include <fastlib.h>
void Histogram (float *a, int ia, float *low, float *high, int
                          nbin,u_long *c, int nr, int nc);
```

Description

The **Histogram** function will calculate the histogram for the float image located at ***a***. The histogram will be placed at ***c.*** Input parameters, ***high*** and ***low,*** provide the function with the range of the input image data. Input parameter, ***nbin,*** specifies the number of bins of the histogram. The following C code fragment describes the function:

```
void Histogram (float *a, int ia, float *low, float *high, int
nbin, u_long *c, int nr, int nc)
{
int i;
int j;
int idx;

       for (i = 0;i < nbin; i ++)
              c[i] = 0;
       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
              {
                     idx = (a[ia*i+j] - *low) / (*high - *low) *
((float)nbin);
                     if (idx < 0)
                            idx = 0;
                     if (idx >= nbin)
                            idx = nbin;
                     c[idx] ++;
              }

}
```

Ifree

Ifree Free dynamically allocated image

C Usage

```
#include <fastlib.h>
void Ifree (void *p);
```

Description

The **Ifree** function frees storage that was dynamically allocated by **Imalloc, Imalloc8** or **Imalloc16**. The argument **p** is the address of an image returned from one of the above functions.

Imalloc

Imalloc Allocate float image

C Usage

```
#include <fastlib.h>
float *Imalloc (int nr, int nc);
```

Description

The **Imalloc** function dynamically allocates a float image of size **nr** by **nc**, returning a float pointer. The value NULL is returned if the allocation fails. The memory space may be freed by calling **Ifree** with the address returned from **Imalloc**.

Imalloc8

Imalloc8 Allocate 8-bit image

C Usage

```
#include <fastlib.h>
unsigned char *Imalloc8 (int nr, int nc);
```

Description

The **Imalloc8** function dynamically allocates an 8-bit image of size **nr** by **nc**, returning an unsigned char pointer. The value NULL is returned if the allocation fails. The memory space may be freed by calling **Ifree** with the address returned from **Imalloc8**.

Imalloc16

Imalloc16 Allocate 16-bit image

C Usage

```
#include <fastlib.h>
unsigned short *Imalloc16 (int nr, int nc);
```

Description

The **Imalloc16** function dynamically allocates a 16-bit image of size **nr** by **nc**, returning an unsigned short pointer. The value NULL is returned if the allocation fails. The memory space may be freed by calling **Ifree** with the address returned from **Imalloc8**.

Interlace

Interlace Convert non-interlaced to interlaced image

C Usage

```
#include <fastlib.h>
void Interlace (unsigned char *a, int ia, unsigned char *odd, int
iodd, unsigned char *even,
int ieven, int nr, int nc);
```

Description

The **Interlace** function takes the non-interlaced image stored in **a**, and breaks it up into odd and even field, storing the result in **odd** and **even**. The argument **nr** specifies the number of rows in a, which is twice the number of rows in **odd** and **even**. The following C code fragment describes the function:

```
void Interlace (unsigned char *a, int ia, unsigned char *odd, int
iodd, unsigned char *even, int ieven, int nr, int nc)
{
       Mov8 (a, 2*ia, odd, iodd, nr, nc);
       Mov8 (a+ia, 2*ia, even, ieven, nr, nc);
}
```

Kirsch

Kirsch Kirsch operator on float image

C Usage

```
#include <fastlib.h>
void Kirsch (float *a, int ia, float *b, int ib, int nrow, int
ncol);
```

Description

The **Kirsch** function applies the kirsch gradient operator to the input image located at ***a***. The gradient image will be placed at ***c***. The following C code fragment describes the function:

```
float   kirsch_h[8][9]
=      {
                              0.33, -0.20, -0.20,
                              0.33,  0.0, -0.20,
                              0.33, -0.20, -0.20,

                              -0.20, -0.20, -0.20,
                              0.33,  0.0, -0.20,
                              0.33,  0.33, -0.20,

                              -0.20, -0.20, -0.20,
                              0.20,  0.0, -0.20,
                              0.33,  0.33,  0.33,

                              -0.20, -0.20, -0.20,
                              0.20,  0.0,  0.33,
                              0.20,  0.33,  0.33,

                              -0.20, -0.20,  0.33,
                              0.20,  0.0,  0.33,
                              0.20, -0.20,  0.33,

                              0.20,  0.33,  0.33,
                              0.20,  0.0,  0.33,
                              0.20, -0.20, -0.20,

                              0.33,  0.33,  0.33,
                              0.20,  0.0, -0.20,
                              0.20, -0.20, -0.20,

                              0.33,  0.33, -0.20,
                              0.33,  0.0, -0.20,
                              0.20, -0.20, -0.20
};

void Kirsch(float *a, int ia, float *b, int ib, int nrow, int
ncol)
{
int row, col, i, j, k, im1, ip1, krn, lcv;
int rowm1xia, rowp1xia, rowxia;
float value, fabs_value, max_val;

       for (row = 1; row < (nrow - 1); row++)
```

```
      {
rowm1xia = (row - 1)*ia;
rowp1xia = (row + 1)*ia;
rowxia = row*ia;
for (col = 1; col < (ncol - 1); col++)
      {
i = rowxia + col;
im1 = rowm1xia + col;
ip1 = rowp1xia + col;
max_val = -999999.0;
for (k = 0; k < 8; k++)
            {
value = kirsch_h[k][0]*a[im1-1];
value += kirsch_h[k][1]*a[im1];
value += kirsch_h[k][2]*a[im1+1];
value += kirsch_h[k][3]*a[i-1];
value += kirsch_h[k][4]*a[i];
value += kirsch_h[k][5]*a[i+1];
value += kirsch_h[k][6]*a[ip1-1];
value += kirsch_h[k][7]*a[ip1];
value += kirsch_h[k][8]*a[ip1+1];

fabs_value = fabs(value);
if (fabs_value > max_val)
                  {
                        max_val = fabs_value;
                  }
            }
j = (ib*row) + col;
b[j] = max_val;
            }
      }
}
```

Lut8

Lut8 Perform lookup on 8-bit image to 8-bit image

C Usage

```
#include <fastlib.h>
void Lut8 (unsigned char *a, int ia, unsigned char *b, unsigned
char *c, int ic, int nr, int nc)
```

Description

The **Lut8** function modifies the 8-bit input image ***(a)*** based on the lookup table, which is passed to the function at ***b***. The 8-bit output image will be placed at ***c***. Table ***b*** contains an intermediate version of the desired lookup table, which is generated by calling **RegLut8. RegLut8** improves performance and must be called once for each lookup table. The following C code fragment describes the function:

```
void Lut8 (unsigned char *a, int ia, unsigned char *b, unsigned
char *c, int ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = b[a[ia*i+j]];
}
```

The following C code fragment demonstrates the use of **RegLut8** with **Lut8**.

```
unsigned char a[512*512], c[512*512];  /* declare image buffers */

unsigned char table_a[256];      /* declare lookup table data */
unsigned char table_b[256];
int lut_a;                /* declare intermediate pointers */
int *lut_b;
initialize (table_a);     /* initialize lookup tables  */
initialize (table_b);
lut_a = RegLut8 (table_a); /* allocate intermediate versions */
lut_b = RegLut8 (table_b);
if (lut_a == NULL || lut_b == NULL) /* check for error */
{
printf ("ERROR: RegLut8 failed\n");
exit (1);
}

/*--- perform lookup using table A ---*/
Lut8 (a, 512, lut_a, c, 512, 512, 512);
/*--- Perform lookup using table B ---*/
Lut8 (a, 512, lut_b, c, 512, 512, 512);
```

Lut8f

Lut8f Perform lookup on 8-bit image to float image

C Usage

```
#include <fastlib.h>
void Lut8f (unsigned char *a, int ia, float *b, float *c, int ic,
int nr,
int nc);
```

Description

The **Lut8f** function modifies the 8-bit input image ***(a)*** based on the lookup table, which is passed to the function at ***b***. The float output image will be placed at ***c***. The following C code fragment describes the function:

```
void Lut8f (unsigned char *a, int ia, float *b, float *c, int ic,
int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = b[a[ia*i+j]];
}
```

Lut8s

Lut8s Perform lookup on 8-bit integer to 16 bit image

C Usage

```
#include <fastlib.h>
void Lut8s (unsigned char *a, int ia, int *b, unsigned short *c,
int ic, int nr,
int nc);
```

Description

The **Lut8s** function modifies the 8-bit input image ***(a)*** based on the 16-bit lookup table, which is passed to the function at ***b***. The 16-bit output image will be placed at ***c***. Table ***b*** contains an intermediate version of the desired lookup table, which is generated by calling **RegLut8s**. **RegLut8s** improves performance and must be called once for each new lookup table. The following C code fragment describes the function:

```
void Lut8s (unsigned char *a, int ia, int *b, unsigned short *c,
int ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
              {
                     c[ic*i+j] = b[a[ia*i+j]];
              }
}
```

Refer to **Lut8** for sample code that demonstrates the use of **RegLut8s** with **Lut8s**.

Lutf

Lutf Perform lookup on float image to float image

C Usage

```
#include <fastlib.h>
void Lutf (float *a, int ia, float *low, float *high, int nbin,
float *b,
float *c, int ic, int nr, int nc);
```

Description

The **Lutf** function modifies the float input image ***(a)*** based on the lookup table which is passed to the function at ***b***. The float output image will be placed at ***c***. Input parameters, ***high*** and ***low,*** provide the function with the range of the input image data. Input parameter, ***nbin,*** specifies the number of bins of the lookup table. The following C code fragment describes the function:

```
void Lutf (float *a, int ia, float *low, float *high, int nbin,
float *b, float *c, int ic, int nr, int nc)
{
int i;
int j;
int idx;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
              {
                     idx = (a[ia*i+j] - *low) / (*high - *low) *
                                                   ((float)nbin);
                     if (idx < 0)
                            idx = 0;
                     if (idx >= nbin)
                            idx = nbin - 1;
                     c[ic*i+j] = b[idx];
              }
}
```

lveq

{xe”lveq”}{xe “blkman”}

Summary

lveq	logical vector equal

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

lveq compares the corresponding elements of real vectors *a* and *b* and sets the corresponding elements of vector *c* to 1.0 if the element of *a* equals the element of *b*, to 0 if not. The processing algorithm is

```
for i=0 to n-1
       if a[i] == b[i],  c[i] = 1.0,
       else c[i] = 0.0
```

lvge

{xe”lvge”}{xe “blkman”}

Summary

lvge	logical vector greater than or equal

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

lvge compares the corresponding elements of real vectors *a* and *b* and if the element of *a* is greater than or equal to the element of *b*, the corresponding element of real vector *c* is set to 1.0. Otherwise, the element of *c* is set to 0.0

```
for i=0 to n-1
       if a[i] >= b[i],  c[i] = 1.0,
       else c[i] = 0.0
```

lvgt

{xe”lvgt”}{xe “blkman”}

Summary

lvgt	logical vector greater than

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

lvge compares the corresponding elements of real vectors *a* and *b*, and if the element of *a* is greater than the element of *b*, the corresponding element of real vector *c* is set to 1.0. Otherwise, the element of *c* is set to 0.0

```
for i=0 to n-1
       if a[i] > b[i],  c[i] = 1.0,
       else c[i] = 0.0
```

lvne

{xe”lvne”}{xe “blkman”}

Summary

lvne	logical vector not equal

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

lvne compares the corresponding elements of real vectors *a* and *b*, and if the element of *a* is not equal to the element of *b*, the corresponding element of real vector *c* is set to 1.0. Otherwise, the element of *c* is set to 0.0

```
for i=0 to n-1
       if a[i] != b[i],  c[i] = 1.0,
       else c[i] = 0.0
```

lvnot

{xe”lvnot”}{xe “blkman”}

Summary

lvnot	logical vector not

C Usage

```
void lvnot (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

lvnot computes the logical complement of real vector *a* and stores the results in real vector *c* according to the algorithm

```
for i=0 to n-1
       if a[i] = 0.0,  c[i] = 1.0,
       else c[i] = 0.0
```

Magnitude

Magnitude Compute magnitude of images

C Usage

```
#include  <fastlib.h>

void Magnitude (float *a, int ia, float *b, int ib, float *c, int
                                                        ic,
             int nr, int nc);
```

Description

Magnitude computes the square root of the sum of the squares of two images. The following C code fragment describes the function:

```
void Magnitude (float *a, int ia, float *b, int ib, float *c, int
ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[i*ic+j] = sqrt (a[i*ia+j]*a[i*ia+j] *
                            b[i*ib+j]*b[i*ib+j]);
}
```

matinv

{xe”matinv”}{xe “blkman”}

Summary

matinv	matrix inverse

C Usage

```
void matinv (float *a, float * c, int nrc, int *ierr);
```

Arguments

a	pointer to real matrix a
c	pointer to resulting inverse matrix c
nrc	number of rows/columns
ierr	pointer to ending condition

Description

matinv computes the inverse of real matrix *a* using the LU decomposition method with back substitution and leaves the inverse in real matrix *c*. The ending condition is indicated in integer *ierr*: a value of zero is returned if the inverse was computed; a value of non zero is returned if the matrix was singular. **matinv** uses temporary working space to hold three rows of data.

Max

Max Maximum of images

C Usage

```
#include <fastlib.h>
void Max (float *a, int ia, float *b, int ib, float *c, int ic,
int nr,
int nc);
```

Description

The **Max** function compares the two input images ***(a and b)*** to find the maximum pixel value at each pixel location. The maximum pixel values create the output image at ***c***. The following C fragment describes the function:

```
#define max(x,y)    ((x) > (y)) ? (x) : (y)
void Max (float *a, int ia, float *b, int ib, float *c, int ic,
int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[i*ic+j] = max (a[i*ia+j],b[i*ib+j]);
}
```

maxmgv

{xe”maxmgv”}{xe “blkman”}

Summary

maxmgv	maximum magnitude of a vector

C Usage

```
void maxmgv (float *a, int ia, float *c, int *lc, int n);
```

Arguments

A	real vector a
ia	stride for vector a
C	return real scalar c containing maximum magnitude found
lc	return integer lc containing index of first element with maximum magnitude
N	element count of vectors

Description

maxmgv finds the first element in real vector *a* having the maximum magnitude (absolute value), stores its magnitude in *c*, and indicates in integer *lc* the number of tests that were performed to reach the first occurrence of the maximum magnitude.

maxv

{xe”maxv”}{xe “blkman”}

Summary

maxv	maximum element of a vector

C Usage

```
void maxv (float *a, int ia, float *c, int *index, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	return real scalar c containing maximum value found
index	return integer Ic containing index of first element with maximum value
n	element count of vectors

Description

maxv finds the element in real vector *a* having the maximum value, stores that value in real scalar *c*, and indicates in integer *index* the number of tests that were performed to reach the first occurrence of the maximum value.

meamgv

{xe”meamgv”}{xe “blkman”}

Summary

meamgv	mean of vector element magnitudes

C Usage

```
void meamgv (float *a, int ia, float *c, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	real scalar c containing computed mean of element magnitudes
n	element count of vectors

Description

meamgv computes the mean magnitude of all elements in real vector *a* and stores the result in real scalar *c*. The algorithm for the mean is:

```
for i=0 to n-1
       c = sum( abs(a[i])) / n
```

meanv

{xe”meanv”}{xe “blkman”}

Summary

meanv	mean value of vector elements

C Usage

```
void meanv (float *a, int ia, float *c, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	real scalar c containing computed mean of vector elements
n	element count of vector a

Description

meanv computes the mean value of all elements in real vector *a* and stores the result in real scalar *c*. The algorithm for the mean is

```
for i=0 to n-1
       c = sum(  a(i)) / N
```

measqv

{xe”measqv”}{xe “blkman”}

Summary

measqv	mean of vector element squares

C Usage

```
void measqv (float *a, int ia, float *c, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	real scalar c containing computed mean of element squares
n	element count of vectors

Description

measqv computes the mean value of the squares of the elements of real vector *a* and stores the result into real scalar *c*. The algorithm for the computation is

```
for i=0 to n-1
       c = sum(a[i] * a[i]) / n
```

Median3x3

Median3x3 3x3 median filter on float image

C Usage

```
#include <fastlib.h>
void Median3x3 (float *a, int ia, float *c, int ic, int nr, int
nc);
```

Description

The **Median3x3** function applies a 3 by 3 median filter to the float input image at ***a***. The output image **©** is created by replacing a pixel in ***a*** with the median value of a nine pixel window centered at the particular pixel. The filtering begins where there is complete coverage of the 3x3 kernel. All border pixels are not filtered and passed directly to the output image.

Median5x5

Median5x5 5x5 median filter on float image

C Usage

```
#include <fastlib.h>
void Median5x5 (float *a, int ia, float *c, int ic, int nr, int
nc);
```

Description

The **Median5x5** function applies a 5 by 5 median filter to the float input image at ***a***. The output image **©** is created by replacing a pixel in ***a*** with the median value of a 25 pixel window centered at the particular pixel. The filtering begins where there is complete coverage of the 5x5 kernel. All border pixels are not filtered and passed directly to the output image.

Median7x7

Median7x7 7x7 median filter on float image

C Usage

```
#include <fastlib.h>
void Median7x7 (float *a, int ia, float *c, int ic, int nr, int
nc);
```

Description

The **Median7x7** function applies a 7 by 7 median filter to the float input image at ***a***. The output image **©** is created by replacing a pixel in ***a*** with the median value of a 49 pixel window centered at the particular pixel. The filtering begins where there is complete coverage of the 7x7 kernel. All border pixels are not filtered and passed directly to the output image.

Min

Min Minimum of images

C Usage

```
#include <fastlib.h>
void Min (float *a, int ia, float *b, int ib, float *c, int ic,
int nr,
int nc);
```

Description

The **Min** function compares the two input images ***(a and b)*** to find the minimum pixel value at each pixel location. The minimum pixel values create the output image at ***c***. The following C fragment describes the function:

```
#define min(x,y)    ((x) < (y)) ? (x) : (y)
void Min (float *a, int ia, float *b, int ib, float *c, int ic,
int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[i*ic+j] = min (a[i*ia+j],b[i*ib+j]);
}
```

minmgv

{xe”minmgv”}{xe “blkman”}

Summary

minmgv	minimum magnitude element of a vector

C Usage

```
void minmgv (float *a, int ia, float *c, int *lc, int n);
```

Arguments

A	real vector a
ia	stride for vector a
C	real scalar c containing minimum magnitude found
lc	integer lc containing index of first element with minimum magnitude
N	element count of vectors

Description

minmgv finds the first element having the minimum magnitude (absolute value) in real vector *a*, stores its magnitude in real scalar *c*, and indicates in integer *lc* the number of tests that were performed to reach the first occurrence of the minimum magnitude.

minv

{xe”minv”}{xe “blkman”}

Summary

minv	minimum element of a vector

C Usage

```
void minv (float *a, int ia, float *c, int *lc, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	real scalar c containing minimum value found
lc	integer lc containing index of first element with minimum value
n	element count of vectors

Description

minv finds the element in real vector *a* having the minimum value, stores that value in real scalar *c*, and indicates in integer *lc* the number of tests that were performed to reach the first occurrence of the minimum value.

mmul

{xe”mmul”}{xe “blkman”}

Summary

mmul	real matrix multiply

C Usage

```
void mmul (float *a, int ia, float * b, int ib, flaot *c, int ic,
int nrc, int ncc, int nca);
```

Arguments

a	real matrix a
ia	stride for matrix a
b	real matrix b
ib	stride for matrix b
c	results in real matrix c
ic	stride for matrix c
nrc	integer number of rows in matrices a and c
ncc	integer number of columns in b and c
nca	integer number of columns in a and rows in b

Description

mmul multiplies the elements of two real matrices *a* and *b* and stores the results in matrix *c*. The strides concept is extended to matrices in this function to specify the address increment between consecutive elements to be processed in the matrices. This is reflected in the manner in which the indices are computed in the algorithm.

```
for i=0 to nrc-1
       for j=0 to ncc-1
              c[i*ncc+j*ic] = 0;
              for k=0 to nca-1
                     c[i*ncc+j*ic] += a[i*nca+k*ia] * b[i*ncc+ib*j]
```

Mov

Mov Move float image

C Usage

```
#include <fastlib.h>
void Mov (float *a, int ia, float *c, int ic, int nr, int nc);
```

Description

The **Mov** function moves the float image at ***a*** to ***c***. The following C fragment describes the function:

```
#include <fastlib.h>
void Mov (float *a, int ia, float *c, int ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i++)
              for (j = 0; j < nc; j++)
                     c[ic*i+j] = a[ia*i+j];
}
```

Mov8

Mov8 Move 8-bit image

C Usage

```
#include <fastlib.h>
void Mov8 (unsigned char *a, int ia, unsigned char *c, int ic, int
nr, int nc);
```

Description

The **Mov8** function moves the 8-bit image at ***a*** to ***c***. The following C fragment describes the function:

```
void Mov8 (unsigned char *a, int ia, unsigned char *c, int ic, int
nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = a[ia*i+j];
}
```

Mov16

Mov16 Move 16 bit image

C Usage

```
#include <fastlib.h>
void Mov16 (unsigned short *a, int ia, unsigned short *c, int ic,
int nr,
int nc);
```

Description

The **Mov16** function moves the 16-bit image at ***a*** to ***c***. The following C fragment describes the function:

```
void Mov16 (unsigned short *a, int ia, unsigned short *c, int ic,
int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = a[ia*i+j];
}
```

mtrans

{xe”mtrans”}{xe “mtrans”}

Summary

mtrans	real matrix transpose

C Usage

```
void mtrans (float *a, int nca, float *c, int ncc, int nraa, int
ncaa);
```

Arguments

a	real input matrix a
nca	integer input number of columns in full matrix a
c	results in real output matrix c
ncc	integer input number of columns in full matrix c
nraa	integer input number of rows in sub matrix a
ncaa	integer input number of columns in sub matrix a

Description

mtrans transposes the elements of a real matrix A and stores the results in C. A and C must not overlay each other.

MulScalar

MulScalar Multiply image by scalar

C Usage

```
#include <fastlib.h>
void MulScalar (float *a, int ia, float *b, float *c, int ic, int
nr,
int nc);
```

Description

The **MulScalar** function multiplies each element of image ***a*** with the scalar given by argument ***b***, placing the output at image ***c***. The following C code fragment describes the function:

```
void MulScalar (float *a, int ia, float *b, float *c, int ic, int
nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i++)
              for (j = 0; j < nc; j++)
                     c[i*ic+j] = a[i*ia+j]* *b;
}
```

Not8

Not8 Compute complement of 8-bit image

C Usage

```
#include <fastlib.h>
void Not8 (unsigned char *a, int ia, unsigned char *c, int ic, int
nr, int nc);
```

Description

The **Not8** function performs a logical complement of each element of image ***a***, placing the output at image ***c***. The following C code fragment describes the function:

```
void Not8 (unsigned char *a, int ia, unsigned char *c, int ic, int
nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = ~a[ia*i+j];
}
```

Not16

Not16 Compute complement of 16 bit image

C Usage

```
#include <fastlib.h>
void Not16 (unsigned short *a, int ia, unsigned short *c, int ic,
int nr,
int nc);
```

Description

The **Not16** function performs a logical complement of each element of image ***a***, placing the output at image ***c***. The following C code fragment describes the function:

```
void Not16 (unsigned short *a, int ia, unsigned short *c, int ic,
int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = ~a[ia*i+j];
}
```

Or8

Or8 Logical OR 8-bit images

C Usage

```
#include <fastlib.h>
void Or8 (unsigned char *a, int ia, unsigned char *b, int ib,
unsigned char *c, int ic, int nr, int nc);
```

Description

The **Or8** function performs a logical “or” of each element of image ***a*** with the corresponding element in image ***b,*** placing the output at image ***c***. The following C code fragment describes the function:

```
void Or8 (unsigned char *a, int ia, unsigned char *b, int ib,
unsigned char *c, int ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = a[ia*i+j] | b[ib*i+j];
}
```

Or16

Or16 Logical OR 16 bit image

C Usage

```
#include <fastlib.h>
void Or16 (unsigned short *a, int ia, unsigned short *b, int ib,
unsigned short *c, int ic, int nr, int nc);
```

Description

The **Or16** function performs a logical “or” of each element of image ***a*** with the corresponding element in image ***b,*** placing the output at image ***c***. The following C code fragment describes the function:

```
void Or16 (unsigned short *a, int ia, unsigned short *b, int ib,
unsigned short *c, int ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = a[ia*i+j] | b[ib*i+j];
}
```

polar

{xe”polar”}{xe “polar”}

Summary

polar	rectangular to polar conversion

C Usage

```
void polar (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	complex vector a
ia	stride for vector a
c	results in complex vector c
ic	stride for vector c
n	element count of vectors

Description

polar converts elements of complex vector *a* from rectangular to polar form and stores the results in complex vector *c*. The algorithm for the conversion is

```
for i=0 to n-1
       real(c[i]) = sqrt(real(a[i])**2 + imag(a[i])**2)
       imag(c[i]) = atan2(imag(a[i]), real(a[i]))
```

PowerSpectrum

PowerSpectrum Power spectrum on float image

C Usage

```
#include <fastlib.h>
void PowerSpectrum(FLOAT *a, int ia, float *c, int ic, int nr, int
nc);
```

Description

The **PowerSpectrum** function determines the power spectrum of the complex FFT located at ***a***. The output power spectrum is placed at ***c***. The following C code fragment describes the function:

```
#include <fastlib.h>
void PowerSpectrum(FLOAT *a, int ia, float *c, int ic, int nr, int
nc)
{
long i, j;
float real, imag;

       for (i = 0; i < nr; i++)
       {
              for (j = 0; j < nc; j++)
              {
real = a[i*ia + j].real;
imag = a[i*ia + j].imag;
c[i*ic + j] = real*real + imag*imag;
              }
       }
}
```

Prewitt

Prewitt Prewitt operator on float image

C Usage

```
#include <fastlib.h>
void Prewitt(float *a, int ia, float *b, int ib, int nrow, int
ncol);
```

Description

The **Prewitt** function applies the Prewitt gradient operator to the input image located at ***a***. The input image is convolved with the two Prewitt operators, which are listed below. The absolute values of the two edge images are summed to give the gradient image. The gradient image is placed at ***c***.

```
float   prewitth1[9] =     {
                                0.33,  0.0,  -0.33,
                                0.33,  0.0,  -0.33,
                                0.33,  0.0,  -0.33
                           );

float   prewitth2[9] = {
                                0.33, -0.33, -0.33,
                                0.00,  0.00,  0.00,
                                0.33,  0.33,  0.33
                           };
```

rect

{xe”rect”}{xe “rect”}

Summary

rect	polar to rectangular conversion

C Usage

```
void rect (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	complex vector a
ia	stride for vector a
c	results in complex vector c
ic	stride for vector c
n	element count of vectors

Description

rect converts the elements of complex vector *a* from polar to rectangular form and stores the results in complex vector *c*. The algorithm for the conversion is

```
for i=0 to n-1
       real(c[i]) = real(a[i]) * cos(imag(a[i]))
       imag(c[i]) = real(a[i]) * sin(imag(a[i]))
```

Reflect

Reflect Reflect float image

C Usage

```
#include <fastlib.h>
void Reflect (float *a, int ia, float *c, int ic, int nr, int nc,
int mode);
```

Description

The **Reflect** function flips the input float image, ***a,*** about the vertical, horizontal or diagonal axis. The axis of reflection is given by the input parameter, ***mode*** (vertical = 0, horizontal = 1, diagonal = 2). The output image is stored at **c**. The following C code fragment describes the function:

```
#define VERTICAL   0
#define HORIZONTAL 1
#define DIAGONAL   2

void Reflect (float *a, int ia, float *c, int ic, int nr, int nc,
int mode)
{
       int i, j, midrow, crow, ccol;
       switch (mode)
       {
       case HORIZONTAL :
              for (i = 0, crow = nc - 1; i < nc; i++, crow--)
                     for (j = 0; j < nc; j++)
                            c[crow*ic + j] = a[i*ia + j];
              break;
       case VERTICAL :
              for (i = 0; i < nr; i++)
                     for (j = 0, ccol = nc - 1; j < nc; j++, ccol--
)
                            c[i*ic + ccol] = a[i*ia + j];
              break;
       case DIAGONAL :
              for (i = 0; i < nr; i++)
                     for (j = 0; j < nc; j++)
                            c[j*ic + i] = a[i*ia + j];
              break;
       default:
              break;
       }
}
```

Reflect8

Reflect8 Reflect 8-bit image

C Usage

```
#include <fastlib.h>
void Reflect8 (unsigned char *a, int ia, unsigned char *c, int ic,
int nr, int nc,
int mode);
```

Description

The **Reflect8** function flips the input 8-bit image, ***a,*** about the vertical, horizontal or diagonal axis. The axis of reflection is given by the input parameter, ***mode*** (vertical = 0, horizontal = 1, diagonal = 2). The output image is stored at **c**. The following C code fragment describes the function:

```
#define VERTICAL   0
#define HORIZONTAL 1
#define DIAGONAL   2

void Reflect8 (unsigned char *a, int ia, unsigned char *c, int ic,
int nr, int nc, int mode)
{
      int i, j, midrow, crow, ccol;
      switch (mode)
      {
      case HORIZONTAL :
            for (i = 0, crow = nc - 1; i < nc; i++, crow--)
                  for (j = 0; j < nc; j++)
                        c[crow*ic + j] = a[i*ia + j];
            break;
      case VERTICAL :
            for (i = 0; i < nr; i++)
                  for (j = 0, ccol = nc - 1; j < nc; j++, ccol--
)
                        c[i*ic + ccol] = a[i*ia + j];
            break;
      case DIAGONAL :
            for (i = 0; i < nr; i++)
                  for (j = 0; j < nc; j++)
                        c[j*ic + i] = a[i*ia + j];
            break;
      default:
            break;
      }
}
```

Reflect16

Reflect16 Reflect 8-bit image

C Usage

```
#include <fastlib.h>
void Reflect16 (unsigned short *a, int ia, unsigned short *c, int
ic,
int nr, int nc, int mode);
```

Description

The **Reflect16** function flips the input 16-bit image, ***a,*** about the vertical, horizontal or diagonal axis. The axis of reflection is given by the input parameter, ***mode*** (vertical = 0, horizontal = 1, diagonal = 2). The output image is stored at **c**. The following C code fragment describes the function:

```
#define VERTICAL   0
#define HORIZONTAL 1
#define DIAGONAL   2

void Reflect16 (unsigned short *a, int ia, unsigned short *c, int
ic, int nr, int nc, int mode)
{
       int i, j, midrow, crow, ccol;
       switch (mode)
       {
       case HORIZONTAL :
              for (i = 0, crow = nc - 1; i < nc; i++, crow--)
                     for (j = 0; j < nc; j++)
                            c[crow*ic + j] = a[i*ia + j];
              break;
       case VERTICAL :
              for (i = 0; i < nr; i++)
                     for (j = 0, ccol = nc - 1; j < nc; j++, ccol--
)
                            c[i*ic + ccol] = a[i*ia + j];
              break;
       case DIAGONAL :
              for (i = 0; i < nr; i++)
                     for (j = 0; j < nc; j++)
                            c[j*ic + i] = a[i*ia + j];
              break;
       default:
              break;
       }
}
```

RegLut8

RegLut8 Allocate 8-Bit Lookup Table

C Usage

```
#include <fastlib.h>
int *RegLut8 (unsigned char *a);
```

Description

The **RegLut8** function allocates and initializes an intermediate form of the 8-bit input lookup table ***a*** suitable for use with **Lut8**. The intermediate table is allocated (using **malloc**) and may be deallocated using free. Multiple intermediate tables may be allocated using **RegLut8**. **RegLut8** returns an integer pointer to the intermediate table, or NULL if the allocation failed.

Refer to **Lut8** for additional information and sample code on the use of **RegLut8**.

RegLut8s

RegLut8s Allocate 16-Bit Lookup Table

C Usage

```
#include <fastlib.h>
int *RegLut8s (unsigned short *a)
```

Description

The **RegLut8s** function allocates and initializes an intermediate form of the 8-bit to 16-bit input lookup table ***a*** suitable for use with **Lut8s**. The intermediate table is allocated (using **malloc**) and may be deallocated using **free**. Multiple intermediate tables may be allocated using **RegLut8s**. **RegLut8s** returns an integer pointer to the intermediate table, or NULL if the allocation failed.

Refer to **Lut8s** for additional information and sample code on the use of **RegLut8s**.

rfft

{xe”rfft”}{xe “rfft”}

Summary

rfft	real to complex FFT (in place)

C Usage

```
void rfft (float *a, int n, int direction)
```

Arguments

a	input vector a.
n	element count of vector. Must be a power of 2.
direction	processing mode 1 for forward fft -1 for inverse fft

Description

rfft computes either a forward real-to-complex or inverse complex-to-real FFT on the data in vector *a*, and stores the results back into vector *a.* When performing a forward fft, vector *a* is a real vector. Vector *a* is a complex vector in packed format when performing an inverse fft.

If *direction* = 1, the function performs a forward FFT. The results are not scaled, and they may be scaled using **rfftsc** to multiply by *1/(2*n)*. if *direction* = -1, the function performs an inverse FFT. The results do not need to be scaled.

A call must have been previously made to the function **bldwts** in order to build a weight vector required by the FFT functions. **bldwts** need only be called once, with an argument specifying the maximum FFT length required in calls to the **rfft** routine. In other words, **bldwts** should be called with the largest argument **n** that will be used with **rrft** during the course of an application.

The **rfft** function uses a standard packed format for complex results of a real FFT. The forward FFT of an *n* element real valued vector produces *n* complex results. But due to symmetry only *(n/2+1)* complex results are independent. The first n/2 complex results are returned in place of the input vector *a*, except that, since symmetry dictates that the imaginary portions of the 0^{th} and n/2th complex results must be zero, the real portion of the n/2th complex result can be (and is) returned in place of the imaginary part of the 0^{th} (first) complex entry.

Rfft2D

Rfft2D 2-D real FFT and inverse on float image

C Usage

```
#include <fastlib.h>

void Rfft2D (float *a, int nr, int nc, int direction);
```

Description

The **Rfft2D** function performs a two dimensional complex FFT on the real data in matrix **a** and stores the results back into matrix **a**. The arguments **nr** and **nc** are the number of rows and columns in image **a**, and both must be powers of 2. If **direction** is 1, a forward FFT is computed, if **direction** is -1, an inverse FFT is performed.

Forward transform results are not scaled, and may multiplied by 1/(2*nr*nc) (refer to MulScalar). Inverse transform results need not be scaled.

A call must have been previously made to the function **bldwts** in order to build a weight vector required by the FFT functions. **bldwts** need only be called once, with an argument specifying the maximum FFT length required in calls to the **rfft2d** routine.

The **Rfft2D** function uses a standard packed format for storing complex results of a real 2D FFT. The forward FFT of an *nr* by *nc* element real valued matrix produces *nr*nc* complex results. But due to symmetry only *nr*nc* real results need be stored. The first two columns of result *a* contain two complex results stored in the **rfft** packed format (refer to **rfft**). The remaining column pairs of *a* form complex results.

The library function **uprft2** may be used to unpack the result of **rfft2d** into an *nr* by *nc* complex matrix, which when scaled produces the same result were the **cfft2d** function used on a real valued complex input matrix.

rfftb

{xe”rfftb”}{xe “rfftb”}

Summary

rfftb	real to complex FFT (not in place)

C Usage

```
void rfftb (float *a, float * c, int n, int direction)
```

Arguments

a	input vector a.
c	output vector c.
n	element count of vector; must be a power of 2
direction	processing mode 1 for forward fft -1 for inverse fft

Description

rfftb computes either a forward or inverse real-to-complex FFT on the data in vector *a*, and stores the results into vector *c.* If a forward FFT is being performed, then vector *a* is a real vector and vector *c* is a complex vector in packed format. If a reverse FFT is being performed, then vector *a* is a complex vector in packed format while vector *c* is a real vector.

If *direction* = 1, the function performs a forward FFT. The results are not scaled, and they may be scaled using **rfftsc** to multiply by *1/(2*n)*. if *direction* = -1, the function performs an inverse FFT. The results to not need to be scaled.

A call must have been previously made to the function **bldwts** in order to build a weight vector required by the FFT functions. **bldwts** need only be called once, with an argument specifying the maximum FFT length required in calls to the **rfftb** routine.

The **rfftb** function uses a standard packed format for complex results of a real FFT. The forward FFT of an *n* element real valued vector produces *n* complex results. But due to symmetry only *(n/2+1)* complex results are independent. The first n/2 complex results are returned in place of the output vector c, except that, since symmetry dictates that the imaginary portions of the 0^{th} and n/2th complex results must be zero, the real portion of the n/2th complex result can be (and is) returned in place of the imaginary part of the 0^{th} (first) complex entry.

rfftsc

{xe”rfftsc”}{xe “rfftsc”}

Summary

rfftsc	real FFT scale and format

C Usage

```
void rfftsc (float *a, int n, int iflag, int iscale)
```

Arguments

a	real input vector a
n	element count of vector
iflag	formatting flag
	iflag = 0, no packing
	iflag = 2, unpack into n/2 complex elements.
	iflag = 3, unpack into n/2+1 complex elements
	iflag = -2, pack from n/2 complex elements to real fft packed format
	iflag = -3, pack from n/2+1 complex elements to real fft packed format
iscale	scale flag
	iscale = 0, no scaling
	iscale = 1, scale elements of a by 1/(2*n)
	iscale = -1, scale elements of a by 1/(4*n)

Description

The **rfftsc** takes the results of a real to complex forward FFT and packs or unpacks it into a more conventional complex format. In addition, the results may be scaled by a factor of *1/(2*n)* or *1/(4*n)*.

The description of packing and unpacking is as follows. Given *A[n/2]* is the complex result of an *n* element real to complex FFT, then:

```
if iflag is 2, then
       real (A[0]) = real (A[0]),
       imag (A[0]) = 0.0

if iflag is 3, then
       real (A[n/2]) = imag (A[0]),
       imag (A[n/2]) = 0.0
       real (A[0]) = real (A[0]),
       imag (A[0]) = 0.0

if iflag is -2, then
       real (A[0]) = real (A[0]),
       imag (A[0]) = 0.0

if iflag is -3, then
       real (A[0]) = real (A[0]),
       imag (A[0]) = real (A[n/2])
```

rmsqv

{xe”rmsqv”}{xe “rmsqv”}

Summary

rmsqv	root mean square of vector elements

C Usage

```
float a[ ], c;
int ia, n;
void rmsqv (float *a, int ia, float *c, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	real scalar c containing result
n	element count of vectors

Description

rmsqv computes the square root of the mean value of the squares of the elements of real vector *a* and stores the result in scalar *c* using the algorithm

```
for i=0 to n-1
       c = sqrt( sum(a[i]*a[i]) / n)
```

Roberts

Roberts Roberts operator on float image

C Usage

```
#include <fastlib.h>
void Roberts (float *a, int ia, float *b, int ib, int nrow, int
ncol);
```

Description

The **Roberts** function applies the Roberts gradient operator to the input image located at ***a***. The input image is convolved with the two Roberts operators, which are listed below. The absolute values of the two edge images are summed to give the gradient image. The gradient image is placed at ***c***.

```
Float         roberts_h1[9] = {
0.0,  0.0, -1.0,
0.0,  1.0,  0.0,
0.0,  0.0,  0.0
                              };
float         roberts_h2[9] = {
•      1.0,  0.0,  0.0,
0.0,  1.0,  0.0,
0.0,  0.0,  0.0
                              };
```

Rotate

Rotate Rotate float image

C Usage

```
#include <fastlib.h>
void Rotate (float *a, int ia, float *b, float *c, int ic, int nr,
int nc);
```

Description

The **Rotate** function performs a rotation of the float image, ***a,*** about the image center. The angle of rotation is given in degrees by ***b**, which rotates the image in a counter-clockwise direction with respect to the x-axis. The output image is stored at **c**. The following C code fragment describes the function:

```
/*
  The input image is rotated by an angle *b about its center in a
counter-   clockwise direction with respect to the x-axis.
  coordinate transformation
  rotation angle will be counter-clockwise  wrt the x-axis
  x-coordinate:
(x' - maxx/2) = (x - maxx/2)*cos(b) - (maxy/2 - y)*sin(b)
x' = x*cos(b) + (1 - cos(b))*maxx/2 - (maxy/2 - y)*sin(b)
xp = x*cos_b  + coef1  + y*sin_b
xp = x*cos_b + ydepend1

  y-coordinate:
(maxy/2 - y') = (x - maxx/2)*sin(b) + (maxy/2 - y)*cos(b)
y' = (maxx/2 - x)*sin(b) + y*cos(b) + (1 - cos(b))*maxy/2
y' = -x*sin(b) + y*cos(b) + sin(b)*nc/2 + (1 - cos(b))*nr/2
yp = -x*sin_b + y*cos_b + coef2
yp = ydepend2 - x*sin_b
*/

void Rotate (float *a, int ia, float *b, float *c, int ic, int nr,
int nc)
{
int x, xp;
int y, yp;
float angle_b, sin_b, nsin_b, cos_b, coef1, coef2;
float ydepend1, ydepend2, nr_shift, nc_shift;

    angle_b = *b * DEG2RAD;
sin_b = (float) sinf(angle_b);
cos_b = (float) cosf(angle_b);

/* negate sine value */
nsin_b = -1.0 * sin_b;

nr_shift = (nr - 1)/2.0;
nc_shift = (nc - 1)/2.0;
coef1 = (1.0 - cos_b)*nc_shift - (nr_shift * sin_b);
coef2 = (1.0 - cos_b)*nr_shift + (nc_shift * sin_b);

    for (y = 0; y < nr; y ++)
    {
ydepend1 = coef1 + y*sin_b;
```

```
ydepend2 = coef2 + y*cos_b;
for (x = 0; x < nc; x ++)
        {
xp = (int)(x*cos_b + ydepend1 + 0.5);
yp = (int)(ydepend2 - x*sin_b + 0.5);
if (xp >= nc || xp < 0 || yp >= nr || yp < 0)
                continue;
            c[ic*yp + xp] = a[ia*y + x];
        }
    }
}
```

Rotate8

Rotate8 Rotate 8-bit image

C Usage

```
#include <fastlib.h>
void Rotate8 (unsigned char *a, int ia, float *b, unsigned char
*c, int ic, int nr,
int nc);
```

Description

The **Rotate** function performs a rotation of the 8-bit image, ***a,*** about the image center. The angle of rotation is given in degrees by ***b,*** which rotates the image in a counter-clockwise direction with respect to the x-axis. The output image is stored at **c**. The following C code fragment describes the function:

```
/*
  The input image is rotated by an angle *b about its center in a
counter-   clockwise direction with respect to the x-axis.
  coordinate transformation
  rotation angle will be counter-clockwise  wrt the x-axis
  x-coordinate:
(x' - maxx/2) = (x - maxx/2)*cos(b) - (maxy/2 - y)*sin(b)
x' = x*cos(b) + (1 - cos(b))*maxx/2 - (maxy/2 - y)*sin(b)
xp = x*cos_b  + coef1  + y*sin_b
xp = x*cos_b + ydepend1

  y-coordinate:
(maxy/2 - y') = (x - maxx/2)*sin(b) + (maxy/2 - y)*cos(b)
y' = (maxx/2 - x)*sin(b) + y*cos(b) + (1 - cos(b))*maxy/2
y' = -x*sin(b) + y*cos(b) + sin(b)*nc/2 + (1 - cos(b))*nr/2
yp = -x*sin_b + y*cos_b + coef2
yp = ydepend2 - x*sin_b
*/

void Rotate8 (unsigned char *a, int ia, float *b, unsigned char
*c, int ic, int nr, int nc)
{
int x, xp;
int y, yp;
float angle_b, sin_b, nsin_b, cos_b, coef1, coef2;
float ydepend1, ydepend2, nr_shift, nc_shift;

    angle_b = *b * DEG2RAD;
sin_b = (float) sinf(angle_b);
cos_b = (float) cosf(angle_b);

/* negate sine value */
nsin_b = -1.0 * sin_b;

nr_shift = (nr - 1)/2.0;
nc_shift = (nc - 1)/2.0;
coef1 = (1.0 - cos_b)*nc_shift - (nr_shift * sin_b);
coef2 = (1.0 - cos_b)*nr_shift + (nc_shift * sin_b);

    for (y = 0; y < nr; y ++)
    {
```

```
ydepend1 = coef1 + y*sin_b;
ydepend2 = coef2 + y*cos_b;
for (x = 0; x < nc; x ++)
        {
xp = (int)(x*cos_b + ydepend1 + 0.5);
yp = (int)(ydepend2 - x*sin_b + 0.5);
if (xp >= nc || xp < 0 || yp >= nr || yp < 0)
                continue;
            c[ic*yp + xp] = a[ia*y + x];
        }
    }
}
```

Rotate16

Rotate16 Rotate 16 bit image

C Usage

```
#include <fastlib.h>
void Rotate16 (unsigned short *a, int ia, float *b, unsigned short
*c,
int ic, int nr, int nc);
```

Description

The **Rotate16** function performs a rotation of the 16-bit image, ***a,*** about the image center. The angle of rotation is given in degrees by ***b,*** which rotates the image in a counter-clockwise direction with respect to the x-axis. The output image is stored at **c**. The following C code fragment describes the function:

```
/*
  The input image is rotated by an angle *b about its center in a
counter-   clockwise direction with respect to the x-axis.
  coordinate transformation
  rotation angle will be counter-clockwise  wrt the x-axis
  x-coordinate:
(x' - maxx/2) = (x - maxx/2)*cos(b) - (maxy/2 - y)*sin(b)
x' = x*cos(b) + (1 - cos(b))*maxx/2 - (maxy/2 - y)*sin(b)
xp = x*cos_b  + coef1  + y*sin_b
xp = x*cos_b + ydepend1

  y-coordinate:
(maxy/2 - y') = (x - maxx/2)*sin(b) + (maxy/2 - y)*cos(b)
y' = (maxx/2 - x)*sin(b) + y*cos(b) + (1 - cos(b))*maxy/2
y' = -x*sin(b) + y*cos(b) + sin(b)*nc/2 + (1 - cos(b))*nr/2
yp = -x*sin_b + y*cos_b + coef2
yp = ydepend2 - x*sin_b
*/

void Rotate16 (unsigned short *a, int ia, float *b, unsigned short
*c, int ic, int nr, int nc)
{
int x, xp;
int y, yp;
float angle_b, sin_b, nsin_b, cos_b, coef1, coef2;
float ydepend1, ydepend2, nr_shift, nc_shift;

    angle_b = *b * DEG2RAD;
sin_b = (float) sinf(angle_b);
cos_b = (float) cosf(angle_b);

/* negate sine value */
nsin_b = -1.0 * sin_b;

nr_shift = (nr - 1)/2.0;
nc_shift = (nc - 1)/2.0;
coef1 = (1.0 - cos_b)*nc_shift - (nr_shift * sin_b);
coef2 = (1.0 - cos_b)*nr_shift + (nc_shift * sin_b);

    for (y = 0; y < nr; y ++)
    {
```

```
ydepend1 = coef1 + y*sin_b;
ydepend2 = coef2 + y*cos_b;
for (x = 0; x < nc; x ++)
        {
xp = (int)(x*cos_b + ydepend1 + 0.5);
yp = (int)(ydepend2 - x*sin_b + 0.5);
if (xp >= nc || xp < 0 || yp >= nr || yp < 0)
                continue;
            c[ic*yp + xp] = a[ia*y + x];
        }
    }
}
```

Sobel

Sobel Sobel operator on float image

C Usage

```
#include <fastlib.h>
void Sobel (float *a, int ia, float *b, int ib, int nrow, int
ncol);
```

Description

The **Sobel** function applies the Sobel gradient operator to the input image located at ***a***. The input image is convolved with the two Sobel operators, which are listed below. The absolute values of the two edge images are summed to give the gradient image. The gradient image is placed at ***c***.

```
float   sobelh1[9] =   {
•      1.00,  0.0,  1.00,
•      2.00,  0.0,  2.00,
•      1.00,  0.0,  1.00
                      };

float   sobelh2[9] =   {
•      1.00, -2.00, -1.00,
                                    0.00,  0.00,  0.00,
                                    1.00,  2.00,  1.00
                      };
```

Sub

Sub Subtract images

C Usage

```
#include <fastlib.h>
void Sub (float *a, int ia, float *b, int ib, float *c, int ic,
int nr,
int nc);
```

Description

The **Sub** function computes the arithmetic difference of two input images. The following C fragment describes the function:

```
void Sub (float *a, int ia, float *b, int ib, float *c, int ic,
int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[i*ic+j] = a[i*ia+j]-b[i*ib+j];
}
```

Sum

Sum Sum float image to float image

C Usage

```
#include <fastlib.h>
void Sum (float *a, int ia, float *c, int ic, int nr, int nc)
```

Description

```
void Sum (float *a, int ia, float *c, int ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] += a[ia*i+j];
}
```

Sum8i

Sum8i Sum 8-bit image to 32 bit image

C Usage

```
#include <fastlib.h>
void Sum8i (unsigned char *a, int ia, u_long *c, int ic, int nr,
int nc);
```

Description

The **Sum8i** function sums the 8-bit image data input from image **a** with the 32 bit image **c**. The following C code fragment describes the function.

```
void Sum8i (unsigned char *a, int ia, u_long *c, int ic, int nr,
int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] += a[ia*i+j];
}
```

Sum16i

Sum16i Sum 16 bit image to 32 bit image

C Usage

```
#include <fastlib.h>
void Sum16i (unsigned short *a, int ia, u_long *c, int ic, int nr,
int nc);
```

Description

The **Sum16i** function sums the 16 bit image data input from image **a** with the 32 bit image **c**. The following C code fragment describes the function.

```
void Sum16i (unsigned short *a, int ia, u_long *c, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] += a[ia*i+j];
}
```

Sum8f

Sum8f Sum 8-bit image to float image

C Usage

```
#include <fastlib.h>
void Sum8f (unsigned char *a, int ia, float *c, int ic, int nr,
int nc);
```

Description

The **Sum8f** function sums the 8-bit image data input from image **a** with float image **c**. The following C code fragment describes the function.

```
void Sum8f (unsigned char *a, int ia, float *c, int ic, int nr,
int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] += (float) a[ia*i+j];
}
```

Sum16f

Sum16f Sum 16 bit image to float image

C Usage

```
#include <fastlib.h>
void Sum16f (unsigned short *a, int ia, float *c, int ic, int nr,
int nc);
```

Description

```
void Sum16f (unsigned short *a, int ia, float *c, int ic, int nr,
int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] += (float) a[ia*i+j];
}
```

svdiv

{xe”svdiv”}{xe “svdiv”}

Summary

svdiv	scalar vector divide

C Usage

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

Arguments

a	real scalar a
b	real vector b
ib	stride for vector b
c	real vector c
ic	stride for vector c
n	element count of vectors

Description

svdiv divides scalar *a* by real vector *b* putting the results in real vector *c*.

```
for i=0 to n-1
       c[i] = a / b[i]
```

sve

{xe”sve”}{xe “sve”}

Summary

sve	sum of vector elements

C Usage

```
void sve (float *a, int ia, float *c, int n);
```

Arguments

A	real vector a
ia	stride for vector a
C	real scalar c containing result
N	element count of vectors

Description

sve stores into real scalar *c* the sum of the elements of real vector *a* computed by the algorithm

```
for i=0 to n-1
       c = sum(a[i])
```

svemg

{xe”svemg”}{xe “svemg”}

Summary

svemg	sum of vector element magnitudes

C Usage

```
void svemg (float *a, int ia, float *c, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	real scalar c containing result
n	element count of vectors

Description

svemg stores into real scalar *c* the sum of the absolute values of the elements of real vector *a* using the algorithm

```
for i=0 to n-1
       c = sum( abs(a[i]) )
```

svesq

{xe”svesq”}{xe “svesq”}

Summary

svesq	sum of vector element squares

C Usage

```
void svesq (float *a, int ia, float *c, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	real scalar c containing result
n	element count of vectors

Description

svesq stores into real scalar *c* the sum the squares of the elements of real vector *a* using the algorithm

```
for i=0 to n-1
       c = sum( a[i]*a[i] )
```

svs

{xe”svs”}{xe “svs”}

Summary

svs	sum of vector signed element squares

C Usage

```
void svs (float *a, int ia, float *c, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	real scalar c containing result
n	element count of vectors

Description

svs stores into real scalar *c* the sum of the signed squares of the elements of real vector *a* using the algorithm

```
for i=0 to n-1
       c = sum( a[i] * abs(a[i]) )
```

Thr8l2m

Thr8l2m Threshold 8-bit to least-to-most packed binary

C Usage

```
#include <fastlib.h>
void Thr8l2m (unsigned char *a, int ia, unsigned char *b, unsigned
char *c, int ic, int nr,
int nc);
```

Description

The **Thr8l2m** function performs an 8-bit thresholding of image **a**, placing the packed binary (ordered least-to-most) in image **c**. The following C code fragment describes the function:

```
static unsigned char bitset[8] = {0x01, 0x02, 0x04, 0x08, 0x10,
0x20, 0x40, 0x80};
void Thr8l2m (unsigned char *a, int ia, unsigned char *b, unsigned
char *c, int ic, int nr, int nc)
{
int i;
int j;
int k;
int threshold;
int bitcnt;

       threshold = *b;
bitcnt = 0;
for (i = 0; i < nr; i ++)
       {
k = 0;
c[ic*i] = 0; /* init to 0 */
for (j = 0; j < nc; j ++)
              {
                     if (a[ia*i + j] >= threshold)
                     {
                            c[ic*i + k] |= bitset[bitcnt];
                     }
bitcnt++;
if (bitcnt > 7)
{  /* begin packing a new byte */
k++;
c[ic*i + k] = 0; /* init to 0 */
bitcnt = 0;
                     }
              }
       }
}
```

Thr8m2l

Thr8m2l Threshold 8-bit to most-to-least packed binary

C Usage

```
#include <fastlib.h>
void Thr8m2l (unsigned char *a, int ia, unsigned char *b, unsigned
char *c, int ic, int nr,
int nc);
```

Description

The **Thr8m2l** function performs an 8-bit thresholding of image **a**, placing the packed binary (ordered most-to-least) in image **c**. The following C code fragment describes the function:

```
static unsigned char bitset[8] = {0x01, 0x02, 0x04, 0x08, 0x10,
0x20, 0x40, 0x80};
void Thr8m2l (unsigned char *a, int ia, unsigned char *b, unsigned
char *c, int ic, int nr, int nc)
{
int i;
int j;
int k;
int threshold;
int bitcnt;

threshold = *b;
bitcnt = 7;
for (i = 0; i < nr; i ++)
      {
k = 0;
c[ic*i] = 0; /* init to 0 */
for (j = 0; j < nc; j ++)
      {
                  if (a[ia*i + j] >= threshold)
                  {
                        c[ic*i + k] |= bitset[bitcnt];
                  }
bitcnt--;
if (bitcnt < 0)
{  /* begin packing a new byte */
k++;
c[ic*i + k] = 0; /* init to 0 */
bitcnt = 7;
                  }
            }
      }
}
```

Thr8

Thr8 Threshold 8-bit

C Usage

```
#include <fastlib.h>
void Thr8 (unsigned char *a, int ia, unsigned char *b, unsigned
char *c, int ic, int nr,
int nc);
```

Description

The **Thr8** thresholds the 8-bit image **a** against the threshold value ***b**. The output generated is 8-bit image **c**, with value decimal 255 for pixels of **a** that equal or exceed the threshold, and 0 for pixels that are less than the threshold. The following C code fragment describes the function:

```
void Thr8 (unsigned char *a, int ia, unsigned char *b, unsigned
char *c, int ic, int nr, int nc)
{
int i;
int j;
int threshold;

       threshold = *b;
       for (i = 0; i < nr; i ++)
       {
              for (j = 0; j < nc; j ++)
              {
                     if (a[ia*i + j] >= threshold)
                            c[ic*i + j] = 255;
                     else
                            c[ic*i + j] = 0;
              }
       }
}
```

Uninterlace

Uninterlace Convert interlaced to non-interlaced image

C Usage

```
#include <fastlib.h>
void Uninterlace (unsigned char *odd, int iodd, unsigned char
*even, int ieven,
unsigned char *c, int ic, int nr, int nc);
```

Description

The **Uninterlace** function takes as input two fields of an interlaced image (**odd** and **even**), and synthesizes a non-interlaced image in **c**. The odd rows of **c** are built from **odd** and the even rows from **even**. The following code describes the function:

```
void Uninterlace (unsigned char *odd, int iodd, unsigned char
*even, int ieven, unsigned char *c, int ic, int nr, int nc)
{
       Mov8 (odd, iodd, c, 2*ic, nr, nc);
       Mov8 (even, ieven, c+ic, 2*ic, nr, nc);
}
```

uprft2

{xe”uprft2”}{xe “svs”}

Summary

uprft2	unpack results of rfft2d

C Usage

```
void uprft2 (float *a, float *c, int nr, int nc);
```

Arguments

a	real input matrix a
c	complex output matrix c
nr	number of matrix rows
nc	number of matrix columns

Description

uprft2 takes the results of the two dimensional real to complex FFT function (**rfft2d**) and unpacks the results from the packed format of **rfft2d** to a full *nr*nc* element complex matrix. The resultant matrix is the same as what would result from using **cfft2d** on a real valued complex input matrix.

vabs

{xe”vabs”}{xe “svs”}

Summary

vabs	vector absolute value

C Usage

```
void vabs (float *a, int ia, float * c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vabs computes the absolute value of the elements of real vector *a* and stores the results into real vector *c*

```
for i=0 to n-1
       c[i] = abs(a[i])
```

vadd

{xe”vadd”}{xe “svs”}

Summary

vadd	vector add

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vadd adds the elements of real vectors *a* and *b* and stores the results into real vector *c*

```
for i=0 to n-1
       c[i] = a[i] + b[i]
```

valog10

{xe”valog10”}{xe “svs”}

Summary

valog10	vector anti-log base 10

C Usage

```
void valog10 (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

valog10 computes the anti-logarithm base 10 of each element of real vector *a* and stores the results in real vector *c* using the algorithm

```
for i=0 to n-1
       c[i] = alog10(a[i])
```

vam

{xe”vam”}{xe “svs”}

Summary

vam	vector add and multiply

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	real vector c
ic	stride for vector c
d	results in real vector d
id	stride for vector d
n	element count of vectors

Description

vam adds elements of real vectors *a* and *b*, multiplies that sum by the corresponding element of real vector *c*, and stores the results into real vector *d*

```
for i=0 to n-1
       d[i] = (a[i] + b[i]) * c[i]
```

vand

{xe”vand”}{xe “svs”}

Summary

vand	vector logical AND

C Usage

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

Arguments

a	integer vector a
ia	stride for vector a
b	integer vector b
ib	stride for vector b
c	results in integer vector c
ic	stride for vector c
n	element count of vectors

Description

vand forms the bitwise logical AND of the corresponding 32-bit integer elements of integer vectors *a* and *b* and stores the results into real vector *c*.

```
for i=0 to n-1
       c[i] = a[i] .AND. b[i]
```

vatan

{xe”vatan”}{xe “svs”}

Summary

vatan	vector arctangent

C Usage

```
void vatan (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in radians in real vector c
ic	stride for vector c
n	element count of vectors

Description

vatan computes the arctangent of the elements of real vector *a* and stores the results in radians in real vector *c* according to the algorithm

```
for i=0 to n-1
       c[i] = atan (a[i])
```

vatan2

{xe”vatan2”}{xe “svs”}

Summary

vatan2	vector two argument arctangent

C Usage

```
void vatan2 (float *a, int ia, float *b, int ib, float *c,
int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	results in radians in real vector c
ic	stride for vector c
n	element count of vectors

Description

vatan2 computes the two argument arctangent of the corresponding elements of real vectors *a* and *b* and stores the results in radians in real vector *c* according to the algorithm

```
for i=0 to n-1
       c[i] = atan2(a[i],b[i])
```

vclip

{xe”vclip”}{xe “svs”}

Summary

Vclip	vector clip

C Usage

```
void vclip (float *a, int ia, float *b, float *c, float *d,
int id, int n);
```

Arguments

a	real vector a
ia	stride for vector a
b	real scalar, lower threshold
c	real scalar, upper threshold
d	results in real vector d
id	stride for vector d
n	element count of vectors

Description

vclip clips the value of each element in real vector *a* to be within the range specified by scalars *b* and *c*. The results are stored in vector *d*. The algorithm is

```
for i=0 to n-1
       if a[i] > c, tmp = c
       else tmp = a[i]
       if tmp < b, tmp = b
       d[i] = tmp
```

vclr

{xe”vclr”}{xe “svs”}

Summary

Vclr	vector clear

C Usage

```
void vclr (float *c, int ic, int n);
```

Arguments

c	results in real vector c
ic	stride for vector c
n	element count of vector c

Description

vclr sets all the elements of real vector *c* to 0.0

```
for i=0 to n-1
       c[i] = 0.0
```

vcos

{xe”vcos”}{xe “svs”}

Summary

Vcos	vector cosine

C Usage

```
void vcos (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a, in radians
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vcos computes the cosine of each element of real vector *a* and stores the results in vector *c*

```
for i=0 to n-1
       c[i] = cos(a[i])
```

vdist

{xe”vdist”}{xe “svs”}

Summary

vdist	vector distance

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vdist computes the square root of the sum of the squares of corresponding elements of real vectors *a* and *b* and stores the results into real vector *c*

```
for i=0 to n-1
       c[i] = sqrt( a[i]*a[i] + b[i]*b[i] )
```

vdiv

{xe”vdiv”}{xe “svs”}

Summary

vdiv	vector divide

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vdiv divides each element of real vector *a* by the corresponding element of real vector *b* and stores the result in real vector *c*. The algorithm is

```
for i=0 to n-1
       c[i] = a[i] / b[i]
```

venvlp

{xe”venvlp”}{xe “svs”}

Summary

venvlp	vector envelope

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	real vector c
ic	stride for vector c
d	results in real vector d
id	stride for vector d
n	element count of vectors

Description

venvlp copies elements of real vector *c* to real vector *d* when the element is outside the range defined by corresponding elements of real vectors *a* and *b*; otherwise it copies 0.0 to the element of vector *d*. The algorithm is written

```
for i=0 to n-1
       if (c[i] > a[i]) d[i] = c[i]
       else if (c[i] < b[i]) d[i] = c[i]
       else d[i] = 0.0
```

veqv

{xe”veqv”}{xe “svs”}

Summary

veqv	vector logical EQUIVALENCE

C Usage

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

Arguments

a	integer vector a
ia	stride for vector a
b	integer vector b
ib	stride for vector b
c	results in integer vector c
ic	stride for vector c
n	element count of vectors

Description

veqv forms the bitwise logical EQUIVALENCE (exclusive NOR) of the corresponding 32-bit integer elements of integer vectors *a* and *b* and stores the results into integer vector *c*

```
for i=0 to n-1
       c[i] = a[i] .XNOR. b[i]
```

vexp

{xe”vexp”}{xe “svs”}

Summary

vexp	vector exponential

C Usage

```
void vexp (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vexp computes the natural exponential of each element of real vector *a* and stores the results in real vector *c* using the algorithm

```
for i=0 to n-1
       c[i] = exp(a[i])
```

vfill

{xe”vfill”}{xe “svs”}

Summary

Vfill	vector fill with constant

C Usage

```
void vfill (float *a, float *c, int ic, int n);
```

Arguments

a	real scalar a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vfill fills the elements of real vector *c* with the real scalar in *a.*

```
for i=0 to n-1
       c[i] = a
```

vfix

{xe”vfix”}{xe “svs”}

Summary

vfix	vector fix to integer

C Usage

```
float a[ ];
int ia, c[ ], ic, n, direction;
void vfix (float *a, int ia, int *c, int ic, int n, int rounding);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in integer vector c
ic	stride for vector c
n	element count of vectors
rounding	rounding mode rounding = 0 to round rounding = 1 to truncate

Description

vfix changes the elements of real vector *a* from type **float** to type **int** and stores the results in integer vector *c*. *rounding* specifies whether to round or truncate

```
for i=0 to n-1
       if rounding=0, c[i] = int( (a[i]+sign(a[i])*0.5) )
       else, c[i] = int( a[i] )
```

vfix8

{xe”vfix8”}{xe “svs”}

Summary

vfix8	vector fix to 8 bit integer

C Usage

```
void vfix8 (float *a, int ia, char *c, int ic, int n, int
rounding);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in 8 bit integer vector c
ic	stride for vector c
n	element count of vectors
rounding	rounding mode rounding = 0 to round rounding = 1 to truncate

Description

vfix8 changes the elements of real vector *a* from type **float** to type **char** (1 byte) and stores the results in integer vector *c*. *rounding* specifies whether to round or truncate.

```
for i=0 to n-1
       if rounding=0, c[i] = short( (a[i]+sign(a[i])*0.5) )
       else, c[i] = short( a[i] )
```

vfix16

Summary

vfix16	vector fix to short integer

C Usage

```
void vfix16 (float *a, int ia, short *c, int ic, int n, int
rounding);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in short integer vector c
ic	stride for vector c
n	element count of vectors
rounding	rounding mode rounding = 0 to round rounding = 1 to truncate

Description

vfix16 changes the elements of real vector *a* from type **float** to type **short** (2 bytes) and stores the results in integer vector *c*. *rounding* specifies whether to round or truncate

```
for i=0 to n-1
       if rounding=0, c[i] = short( (a[i]+sign(a[i])*0.5) )
       else, c[i] = short( a[i] )
```

vfix32

{xe”vfix32”}{xe “svs”}

Summary

vfix32	vector fix to long integer

C Usage

```
void vfix32 (float *a, int ia, long *c, int ic, int n, int
rounding);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in long integer vector c
ic	stride for vector c
n	element count of vectors
rounding	rounding mode rounding = 0 to round rounding = 1 to truncate

Description

vfix16 changes the elements of real vector *a* from type **float** to type **long** (4 bytes) and stores the results in integer vector *c*. *rounding* specifies whether to round or truncate

```
for i=0 to n-1
       if rounding=0, c[i] = long( (a[i]+sign(a[i])*0.5) )
       else, c[i] = long( a[i] )
```

vflt

{xe”vflt”}{xe “svs”}

Summary

vflt	vector float

C Usage

```
void vflt (int *a, int ia, float *c, int ic, int n);
```

Arguments

a	integer vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vflt converts the elements of integer vector *a* from type **integer** to type **float** and stores the results in real vector *c*

```
for i=0 to n-1
       c[i] = float(a[i])
```

vflt8

{xe”vflt8”}{xe “svs”}

Summary

vflt8	vector float byte integers

C Usage

```
void vflt8 (char *a, int ia, float *c, int ic, int n);
```

Arguments

a	signed char vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vflt8 converts the elements of signed char vector *a* from type **char** to type **float** and stores the results in real vector *c*

```
for i=0 to n-1
       c[i] = float(a[i])
```

vflt16

{xe”vfild16”}{xe “svs”}

Summary

vflt16	vector float short integers

C Usage

```
void vflt16 (short *a, int ia, float *c, int ic, int n);
```

Arguments

a	short integer vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vflt16 converts the elements of short integer vector *a* from type **short** to type **float** and stores the results in real vector *c*

```
for i=0 to n-1
       c[i] = float(a[i])
```

vflt32

{xe”vfl32”}{xe “svs”}

Summary

vflt32	vector float short integers

C Usage

```
void vflt32 (unsigned int *a, int ia, float *c, int ic, int n);
```

Arguments

a	integer vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vflt32 converts the elements of integer vector *a* from type **unsigned int** to type **float** and stores the results in real vector *c*

```
for i=0 to n-1
       c[i] = float(a[i])
```

vfltu8

{xe”vfltu8”}{xe “svs”}

Summary

vfltu8	vector float unsigned char integers

C Usage

```
void vfltu8 (unsigned char *a, int ia, float *c, int ic, int n);
```

Arguments

a	unsigned char vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vfltu8 converts the elements of unsigned char vector *a* from type **unsigned char** to type **float** and stores the results in real vector *c*

```
for i=0 to n-1
       c[i] = float(a[i])
```

vfltu16

{xe”vflut6”}{xe “svs”}

Summary

vfltu16	vector float unsigned short integers

C Usage

```
void vfltu16 (unsigned short*a, int ia, float *c, int ic, int n);
```

Arguments

a	short integer vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vfltu16 converts the elements of unsigned short integer vector *a* from type **unsigned short** to type **float** and stores the results in real vector *c*

```
for i=0 to n-1
       c[i] = float(a[i])
```

vfltu32

{xe”vfltu32”}{xe “svs”}

Summary

vfltu32	vector float unsigned short integers

C Usage

```
void vfltu32 (unsigned int *a, int ia, float *c, int ic, int n);
```

Arguments

a	unsigned integer vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vfltu32 converts the elements of unsigned integer vector *a* from type **unsigned int** to type **float** and stores the results in real vector *c*

```
for i=0 to n-1
       c[i] = float(a[i])
```

vfrac

{xe”vfrac”}{xe “vfrac”}

Summary

vfrac	vector truncate to fraction

C Usage

```
void vfrac (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vfrac extracts the fractional part of each element of real vector *a* and stores it in the corresponding element of real vector *c* using the algorithm

```
for i=0 to n-1
       c[i] = a[i] - float( int(a[i]) )
```

vgathr

{xe”vgathr”}{xe “vgathr”}

Summary

vgathr	vector gather

C Usage

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

Arguments

a	real vector a of values
b	real vector b of indices
ib	stride for vector b
c	result in real vector c
ic	stride for vector c
n	element count of vectors

Description

vgathr uses the elements of integer vector *b* as the indices (in the range 0 to *n*-1) by which to fetch the elements of real vector *a* for storage into real vector *c*. No check is made on the validity of the indices in vector *b*.

```
for i=0 to n-1
      c[i] = a[b[i]]
```

vimag

{xe”vimag”}{xe “vimag”}

Summary

vimag	extract imaginaries of complex vector

C Usage

```
void vimag (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	complex vector a
ia	stride for vector a expressed in floats
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vimag forms a real vector *c* from the imaginary parts of complex vector *a*

```
for i=0 to n-1
       c[i] = imag(a[i])
```

Note that the *stride* argument(s) to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

vlim

{xe”vlim”}{xe “vlim”}

Summary

Vlim	vector limit

C Usage

```
void vlim (float *a, int ia, float *b, float *c, float *d,
int id, int n);
```

Arguments

a	real vector a
ia	stride for vector a
b	real scalar threshold
c	real scalar replacement value
d	results in real vector d
id	stride for vector d
n	element count of vectors

Description

vlim creates a real vector *d* with values of only real value *c* or *-c* depending on whether the corresponding element of real vector *a* is less than the threshold value *b* using the algorithm

```
for i=0 to n-1
       if a[i] < b, d[i] = -c,
       else, d[i] = c
```

vlog

{xe”vlog”}{xe “vlog”}

Summary

vlog	vector natural logarithm

C Usage

```
void vlog (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vlog stores the natural logarithm of the elements of real vector *a* into real vector *c*

```
for i=0 to n-1
       c[i] = log(a[i])
```

vlog10

{xe”vlog10”}{xe “vlog10”}

Summary

vlog10	vector base 10 logarithm

C Usage

```
void vlog10 (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vlog10 stores the base 10 logarithm of the elements of real vector *a* into real vector *c*

```
for i=0 to n-1
       c[i] = log10(a[i])
```

vma

{xe”vma”}{xe “vma”}

Summary

vma	vector multiply and add

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	real vector c
ic	stride for vector c
d	results in real vector d
id	stride for vector d
n	element count of vectors

Description

vma multiplies elements of real vectors *a* and *b*, adds that product to the corresponding element of real vector *c*, and stores the result into real vector *d* using the algorithm

```
for i=0 to n-1
       d[i] = (a[i] * b[i]) + c[i]
```

vmax

{xe”vmax”}{xe “vmax”}

Summary

vmax	vector maximum of two vectors

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vmax select the maximum of corresponding elements of real vectors *a* and *b* and stores the result in real vector *c*

```
for i=0 to n-1
       c[i] = max( a[i], b[i] )
```

vmaxmg

{xe”vmaxmg”}{xe “vmaxmg”}

Summary

vmaxmg	vector maximum magnitude of two vectors

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vmaxmg select the maximum of the absolute values of corresponding elements of real vectors *a* and *b* and stores the result in real vector *c*

```
for i=0 to n-1
       c[i] = max( abs(a[i]), abs(b[i]) )
```

vmin

{xe”vmin”}{xe “vmin”}

Summary

vmin	vector minimum of two vectors

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vmin selects the minimum values of corresponding elements of real vectors *a* and *b* and stores the result in real vector *c*

```
for i=0 to n-1
       c[i] = min( a[i], b[i] )
```

vminmg

{xe”vminmg”}{xe “vminmg”}

Summary

vminmg	vector minimum magnitude of two vectors

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vminmg select the minimum of the absolute values of corresponding elements of real vectors *a* and *b* and stores the result in real vector *c*

```
for i=0 to n-1
       c[i] = min( abs(a[i]), abs(b[i]) )
```

vmov

{xe”vmov”}{xe “vmov”}

Summary

vmov vector move

C Usage

```
void vmov (float *a, int ia, float *c, int ic, int n);
```

Arguments

A	real vector A
IA	stride for vector A
C	results in real vector C
IC	stride for vector C
N	element count of vectors

Description

vmov copies the elements of real vector *a* to real vector *c*

```
for i=0 to n-1
       c[i] = a[i]
```

vmsa

{xe”vmsa”}{xe “vmsa”}

Summary

vmsa	vector multiply and scalar add

C Usage

```
void vmsa (float *a, int ia, float *b, int ib, float *c,
float *d, int id, int n);
```

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	real scalar c
d	results in real vector d
id	stride for vector d
n	element count of vectors

Description

vmsa multiplies elements of real vectors *a* and *b*, adds that product to real scalar *c*, and stores the result into real vector *d* using the algorithm

```
for i=0 to n-1
       d[i] = (a[i] * b[i]) + c
```

vmsb

{xe”vmsb”}{xe “vmsb”}

Summary

vmsb	vector multiply and subtract

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	real vector c
ic	stride for vector c
d	results in real vector d
id	stride for vector d
n	element count of vectors

Description

vmsb multiplies elements of real vectors *a* and *b*, subtracts from that product the corresponding element of real vector *c*, and stores the result into real vector *d* using the algorithm

```
for i=0 to n-1
       d[i] = (a[i] * b[i]) - c[i]
```

vmul

{xe”vmul”}{xe “vmul”}

Summary

vmul	vector multiply

C Usage

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

Arguments

A	real vector a
ia	stride for vector a
B	real vector b
ib	stride for vector b
C	results in real vector c
ic	stride for vector c
N	element count of vectors

Description

vmul multiplies the corresponding elements of real vectors *a* and *b* and stores the results into real vector *c*

```
for i=0 to n-1
       c[i] = a[i] * b[i]
```

vnabs

{xe”vnabs”}{xe “vnabs”}

Summary

vnabs	vector negative absolute value

C Usage

```
void vnabs (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vnabs stores the negative absolute values of the elements of real vector *a* into real vector *c*

```
for i=0 to n-1
       c[i] = -abs(a[i])
```

vneg

{xe”vneg”}{xe “vneg”}

Summary

vneg	vector negate

C Usage

```
void vneg (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vneg stores the negative of the elements of real vector *a* into real vector *c*

```
for i=0 to n-1
       c[i] = -a[i]
```

vor

{xe”vor”}{xe “vor”}

Summary

vor	vector logical OR

C Usage

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

Arguments

a	integer vector a
ia	stride for vector a
b	integer vector b
ib	stride for vector b
c	results in integer vector c
ic	stride for vector c
n	element count of vectors

Description

vor forms the bitwise logical OR of the corresponding 32-bit integer elements of integer vectors *a* and *b* and stores the results into real vector *c*

```
for i=0 to n-1
       c[i] = a[i] .OR. b[i]
```

vpoly

{xe”vpoly”}{xe “vpoly”}

Summary

vpoly	vector polynomial evaluation

C Usage

```
float a[ ], b[ ], c[ ];
int ia, ib, ic, n, m;
void vpoly (float *a, int ia, float *b, int ib, float *c, int ic,
int n, int order);
```

Arguments

a	real vector a containing polynomial coefficients
ia	stride for vector a
b	real vector b containing independent variable
ib	stride for vector b
c	results in real vector c
ic	stride for vector c
n	element count of vectors
order	integer order of polynomial

Description

vpoly evaluates the polynomial whose coefficients are provided in real vector *a* for each element of real vector *b* used as the independent variable and stores the results into real vector *c*. The coefficients are arranged in descending order in vector *a*. Argument *order* specifies the order of the polynomial stored in vector *a* and must be greater than or equal to 0. The evaluation algorithm is:

```
for i=0 to n-1
       for j=0 to order
              c[i] = sum( a[j] * b[i]**(order-j) )
```

vramp

{xe”vramp”}{xe “vramp”}

Summary

vramp	vector fill with ramp

C Usage

```
void vramp (float *a, float *b, float *c, int ic, int n);
```

Arguments

a	real scalar a, the initial ramp value
b	real scalar b, the ramp increment
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vramp stores the ramp specified by real scalars *a* and *b* into real vector *c* using the algorithm

```
for i=0 to n-1
       c[i] = a + (i * b)
```

vrcip

{xe”vrcip”}{xe “ac”}

Summary

vrcip	vector reciprocal

C Usage

```
void vrcip (float *a, int ia, float *c, int ic, int n);
```

Arguments

A	complex vector A
IA	stride for vector A
C	results in real vector C
IC	stride for vector C
N	element count of vectors

Description

vrcip computes the reciprocal of real vector *a* putting the result in real vector *c.*

```
for i=0 to n-1
       c[i] = 1.0 / a[i]
```

vreal

{xe”vreal”}{xe “vreal”}

Summary

Vreal	extract reals of complex vector

C Usage

```
void vreal (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	complex vector a
ia	stride for vector a expressed in floats
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vreal copies the real parts of complex vector *a* to real vector *c*

```
for i=0 to n-1
       c[i] = real(a[i])
```

Note that the *stride* argument(s) to complex vectors are expressed in floats or reals. To process a contiguous complex array, stride should be 2. To skip every other complex element, stride should be 4.

vsadd

{xe”vsadd”}{xe “vsadd”}

Summary

Vsadd	vector scalar add

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real scalar b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vsadd adds scalar *b* to each element of real vector *a* and stores the results into real vector *c*

```
for i=0 to n-1
       c[i] = a[i] + b
```

vsbm

{xe”vsbm”}{xe “vsbm”}

Summary

vsbm	vector subtract and multiply

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	real vector c
ic	stride for vector c
d	results in real vector d
id	stride for vector d
n	element count of vectors

Description

vsbm subtracts corresponding elements of real vector *b* from those of real vector *a*, multiplies that difference by the corresponding element of real vector *c*, and stores the result into real vector *d* using the algorithm

```
for i=0 to n-1
       d[i] = (a[i] - b[i]) * c[i])
```

vscal

{xe”vscal”}{xe “vscal”}

Summary

vscal	vector scale and fix

C Usage

```
void vscal (float *a, int ia, float *b, int *c, int ic,
int n, int nbits);
```

Arguments

a	real vector a
ia	stride for vector a
b	real scalar b
c	results in integer vector c
ic	stride for vector c
n	element count of vectors
nbits	number of bits

Description

vscal scales the elements of real vector *a* by a power of 2, fixes the scaled values truncating towards 0.0, and stores the results into integer vector *c*. The power of 2 is chosen so that real scalar *b* falls within one-quarter to one-half the dynamic range specified by integer bit width *nbits*

```
k = nbits - int (log2(b))
for i=0 to n-1
       c[i] = fix (a[i]*(2^k))
```

vscatr

{xe”vscatr”}{xe “vscatr”}

Summary

vscatr	vector scatter

C Usage

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

Arguments

A	real vector a of values
ia	stride for vector a
B	integer vector b of indices
ib	stride for vector b
C	results in real vector c
N	element count of vectors

Description

vscatr fetches elements from real vector *a* and stores those elements into vector *c* using indices from integer vector *b* to specify locations in vector *c* in which to store

```
for i=0 to n-1
       c[b[i]] = a[i]
```

vsdiv

{xe”vsdiv”}{xe “vsdiv”}

Summary

vsdiv	vector scalar divide

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real scalar b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vsdiv divides the elements of real vector *a* by scalar *b* and stores the result into real vector *c*

```
for i=0 to n-1
       c[i] = a[i] / b
```

vsin

{xe”vsin”}{xe “vsin”}

Summary

vsin	vector sine

C Usage

```
void vsin (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vsin stores the sine of the elements of real vector *a* into to real vector *c:*

```
for i=0 to n-1
       c[i] = sin(a[i])
```

vsma

{xe”vsma”}{xe “vsma”}

Summary

Vsma	vector scalar multiply and add

C Usage

```
void vsma (float *a, int ia, float *b, float *c, int ic,
float *d, int id, int n);
```

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
c	real vector c
ic	stride for vector c
d	results in real vector d
id	stride for vector d
n	element count of vectors

Description

vsma multiplies elements of real vector *a* by scalar *b*, adds the corresponding element of real vector *c*, and stores the result into real vector *d* using the algorithm

```
for i=0 to n-1
       d[i] = (a[i] * b) + c[i]
```

vsmb

{xe”vsmb”}{xe “vsmb”}

Summary

vsma	vector scalar multiply and subtract

C Usage

```
void vsmb (float *a, int ia, float *b, float *c, int ic,
float *d, int id, int n);
```

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
c	real vector c
ic	stride for vector c
d	results in real vector d
id	stride for vector d
n	element count of vectors

Description

vsmb multiplies elements of real vector *a* by scalar *b*, subtracts the corresponding element of real vector *c*, and stores the result into real vector *d* using the algorithm

```
for i=0 to n-1
       d[i] = (a[i] * b) - c[i]
```

vsmsa

{xe”vsmsa”}{xe “vsmsa”}

Summary

vsmsa	vector scalar multiply and scalar add

C Usage

```
void vsmsa (float *a, int ia, float *b, float *c, float *d, int
id,
int n);
```

Arguments

a	real vector a
ia	stride for vector a
b	real scalar b
c	real scalar c
d	results in real vector d
id	stride for vector d
n	element count of vectors

Description

vsmsa multiplies elements of real vector *a* by scalar *b*, adds real scalar *c*, and stores the result into real vector *d* using the algorithm

```
for i=0 to n-1
       d[i] = (a[i] * b) + c
```

vsmsb

{xe”vsmab”}{xe “vsmsb”}

Summary

Vsmsb	vector scalar multiply and subtract

C Usage

```
void vsmsb (float *a, int ia, float *b, float *c, int ic,
float *d, int id, int n);
```

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	real vector c
ic	stride for vector c
d	results in real vector d
id	stride for vector d
n	element count of vectors

Description

vsmsb multiplies elements of real vector *a* by scalar *b*, subtracts the corresponding element of real vector *c*, and stores the result into real vector *d* using the algorithm

```
for i=0 to n-1
       d[i] = (a[i] * b) - c[i]
```

vsmul

{xe”vsmul”}{xe “vsmul”}

Summary

vsmul	vector scalar multiply

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real scalar b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vsmul multiplies the elements of real vector *a* by scalar *b* and stores the result into real vector *c*

```
for i=0 to n-1
       c[i] = a[i] * b
```

vsq

{xe”vsq”}{xe “vsq”}

Summary

vsq	vector square

C Usage

```
void vsq (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vsq stores the square of each element of real vector *a* into to real vector *c*

```
for i=0 to n-1
       c[i] = a[i] * a[i]
```

vsqrt

{xe”vsqrt”}{xe “vsqrt”}

Summary

vsqrt	vector square root

C Usage

```
void vsqrt (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vsqrt stores the square root of each element of real vector *a* into to real vector *c*

```
for i=0 to n-1
       c[i] = sqrt(a[i])
```

vssq

{xe”vssq”}{xe “vssq”}

Summary

vssq	vector signed square

C Usage

```
void vssq (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vssq multiplies each element of real vector *a* by the absolute value of the element and stores the result into to real vector *c*

```
for i=0 to n-1
       c[i] = a[i] * abs(a[i])
```

vsub

{xe”vsub”}{xe “vsub”}

Summary

vsub	vector subtract

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real vector b
ib	stride for vector b
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vsub subtracts the corresponding elements of real vector *b* from those of real vector *a* and stores the result in real vector *c*

```
for i=0 to n-1
       c[i] = a[i] - b[i]
```

vswap

{xe”vfrac”}{xe “vswap”}

Summary

vswap	vector swap

C Usage

```
void vswap (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vswap swaps the elements of real vector *a* with the corresponding elements of real vector *c*

```
for i=0 to n-1
       temp = c[i], c[i] = a[i], a[i] = temp
```

vtan

{xe”vtan”}{xe “vtan”}

Summary

vtan	vector tangent

C Usage

```
void vtan (float *a, int ia, float *c, int ic, int n);
```

Arguments

a	real vector a in radians
ia	stride for vector a
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vtan computes the tangent of each elements of real vector *a* (in radians) and stores the result into real vector *c*

```
for i=0 to n-1
       c[i] = tan(a[i])
```

vthresh

{xe”vthresh”}{xe “vthresh”}

Summary

Vthresh	vector threshold
Vthr	alias for vector threshold

C Usage

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

Arguments

a	real vector a
ia	stride for vector a
b	real scalar b, lower threshold
c	results in real vector c
ic	stride for vector c
n	element count of vectors

Description

vthresh limits the value of the elements of real vector *a* to be greater than or equal to the threshold in scalar *b* and stores the result into real vector *c.*

```
for i=0 to n-1
       if a[i] < b, c[i] = b
       else, c[i] = a[i]
```

vthr is an allowed alias for **vthresh**.

Xgradient

Xgradient X gradient operator on float data

C Usage

```
#include <fastlib.h>
void Xgradient (float *a, int ia, float *c, int ic, int nrow, int
ncol);
```

Description

The **Xgradient** function convolves a row gradient operator (listed below) with the input image located at ***a***. The X gradient image is placed at ***c***.

```
float    xgrad_h1[9] = {
                            0.0,  0.0,  0.0,
                            0.0,  1.0, -1.0,
                            0.0,  0.0,  0.0
                        };
```

Xor8

Xor8 Logical XOR 8-bit images

C Usage

```
#include <fastlib.h>
void Xor8 (unsigned char *a, int ia, unsigned char *b, int ib,
                    unsigned char *c, int ic, int nr, int nc);
```

Description

The **Xor8** function performs the exclusive-or function between 8-bit images **a** and **b**, placing the result in image **c**. The following C code fragment describes the function:

```
void Xor8 (unsigned char *a, int ia, unsigned char *b, int ib,
unsigned char *c, int ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = a[ia*i+j] ^ b[ib*i+j];
}
```

Xor16

Xor16 Logical XOR 16 bit images

C Usage

```
#include <fastlib.h>
void Xor16 (unsigned short *a, int ia, unsigned short *b, int ib,
unsigned short *c, int ic, int nr, int nc);
```

Description

The **Xor16** function performs the exclusive-or function between 16 bit images **a** and **b**, placing the result in image **c**. The following C code fragment describes the function:

```
void Xor16 (unsigned short *a, int ia, unsigned short *b, int ib,
unsigned short *c, int ic, int nr, int nc)
{
int i;
int j;

       for (i = 0; i < nr; i ++)
              for (j = 0; j < nc; j ++)
                     c[ic*i+j] = a[ia*i+j] ^ b[ib*i+j];
}
```

Ygradient

Ygradient Y gradient operator on float data

C Usage

```
#include <fastlib.h>
void Ygradient(float *a, int ia, float *c, int ic, int nrow, int
ncol);
```

Description

The **Ygradient** function convolves a column gradient operator (listed below) with the input image located at ***a***. The Y gradient image is placed at ***c***.

```
float    ygrad_h1[9] = {
                           0.0, -1.0,  0.0,
                           0.0,  1.0,  0.0,
                           0.0,  0.0,  0.0
                       };
```

Zoom

Zoom Zoom float image

C Usage

```
#include <fastlib.h>
void Zoom (float *a, int ia, float *c, int ic, int nra, int nca,
                                                  int nrc, int ncc);
```

Description

The **Zoom** function expands or shrinks float image **a** of size **nra** by **nca**, to float image **c** of size **nrc** by **ncc** using bi-linear interpolation.

Zoom8

Zoom8 Zoom 8-bit image

C Usage

```
#include <fastlib.h>
void Zoom8 (unsigned char *a, int ia, unsigned char *c, int ic,
int nra, int nca,
int nrc, int ncc);
```

Description

The **Zoom8** function expands or shrinks 8-bit image **a** of size **nra** by **nca**, to 8-bit image **c** of size **nrc** by **ncc** using bi-linear interpolation.

Zoom16

Zoom16 Zoom 16 bit image

C Usage

```
#include <fastlib.h>
void Zoom16 (unsigned short *a, int ia, unsigned short *c, int ic,
                          int nra, int nca, int nrc, int ncc);
```

Description

The **Zoom8** function expands or shrinks 16 bit image **a** of size **nra** by **nca**, to 16 bit image **c** of size **nrc** by **ncc** using bi-linear interpolation.

III. TROUBLESHOOTING

There are several things you can try before you call Alacron Technical Support for help.

_____ Make sure the computer is plugged in. Make sure the power source is on.

_____ Go back over the hardware installation to make sure you didn't miss a page or a section.

_____ Go back over the software installation to make sure you have installed all necessary software.

_____ Run the Installation User Test to verify correct installation of both hardware and software.

_____ Run the user-diagnostics test for your main board to make sure it's working properly.

_____ Insert the Alacron CD-ROM and check the various Release Notes to see if there is any information relevant to the problem you are experiencing.

The release notes are available in the directory: **\usr\alacron\alinfo**

_____ Compile and run the example programs found in the directory: **\usr\alacron\src\examples**

_____ Find the appropriate section of the Programmer's Guide & Reference or the Library User's Manual for the particular library and problem you are experiencing. Go back over the steps in the guide.

_____ Check the programming examples supplied with the runtime software to see if you are using the software according to the examples.

_____ Review the return status from functions and any input arguments.

_____ Simplify the program as much as possible until you can isolate the problem. Turning off any operations not directly related may help isolate the problem.

_____ Finally, first **save your original work**. Then remove any extraneous code that doesn't directly contribute to the problem or failure.

IV. ALACRON TECHNICAL SUPPORT

Alacron offers technical support to any licensed user during the normal business hours of 9 a.m. to 5 p.m. EST. We offer assistance on all aspects of processor board and PMC installation and operation.

A. Contacting Technical Support

To speak with a Technical Support Representative on the telephone, call the number below and ask for Technical Support:

Telephone: **603-891-2750**

If you would rather FAX a written description of the problem, make sure you address the FAX to Technical Support and send it to:

Fax: **603-891-2745**

You can email a description of the problem to ***support@alacron.com***

Before you contact technical support have the following information ready:

_____ Serial numbers and hardware revision numbers of all of your boards. This information is written on the invoice that was shipped with your products.

_____ Also, each board has its serial number and revision number written on either in ink or in bar-code form.

_____ The version of the ALRT, ALFAST, or FASTLIB software that you are using.

_____ You can find this information in a file in the directory: **\usr\alfast\alinfo**

_____ The type and version of the host operating system, i.e., Windows 98.

_____ Note the types and numbers of all your software revisions, daughter card libraries, the application library and the compiler

_____ The piece of code that exhibits the problem, if applicable. If you email Alacron the piece of code, our Technical-Support team can try to reproduce the error. It is necessary, though, for all the information listed above to be included, so Technical Support can duplicate your hardware and system environment.

B. Returning Products for Repair or Replacements

Our first concern is that you be pleased with your Alacron products.

If, after trying everything you can do yourself, and after contacting Alacron Technical Support, you feel your hardware or software is not functioning properly, you can return the product to Alacron for service or replacement. Service or replacement may be covered by your warranty, depending upon your warranty.

The first step is to call Alacron and request a "Return Materials Authorization" (RMA) number.

This is the number assigned both to your returning product and to all records of your communications with Technical Support. When an Alacron technician receives your returned hardware or software he will match its RMA number to the on-file information you have given us, so he can solve the problem you've cited.

When calling for an RMA number, please have the following information ready:

_____ Serial numbers and descriptions of product(s) being shipped back

_____ A listing including revision numbers for all software, libraries, applications, daughter cards, etc.

_____ A clear and detailed description of the problem and when it occurs

_____ Exact code that will cause the failure

_____ A description of any environmental condition that can cause the problem

All of this information will be logged into the RMA report so it's there for the technician when your product arrives at Alacron.

Put boards inside their anti-static protective bags. Then pack the product(s) securely in the original shipping materials, if possible, and ship to:

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Clearly mark ***the outside of your package:***

Attention **RMA #80XXX**

Remember to include your return address and the name and number of the person who should be contacted if we have questions.

C. Reporting Bugs

We at Alacron are continually improving our products to ensure the success of your projects. In addition to ongoing improvements, every Alacron product is put through extensive and varied testing. Even so, occasionally situations can come up in the fields that were not encountered during our testing at Alacron.

If you encounter a software or hardware problem or anomaly, please contact us immediately for assistance. If a fix is not available right away, often we can devise a work-around that allows you to move forward with your project while we continue to work on the problem you've encountered.

It is important that we are able to reproduce your error in an isolated test case. You can help if you create a stand-alone code module that is isolated from your application and yet clearly demonstrates the anomaly or flaw.

Describe the error that occurs with the particular code module and email the file to us at:

support@alacron.com

We will compile and run the module to track down the anomaly you've found.

If you do not have Internet access, or if it is inconvenient for you to get to access, copy the code to a disk, describe the error, and mail the disk to Technical Support at the Alacron address below.

If the code is small enough, you can also:

FAX the code module to us at 603-891-2745

If you are faxing the code, write everything large and legibly and remember to include your description of the error.

When you are describing a software problem, include revision numbers of all associated software.

For documentation errors, photocopy the passages in question, mark on the page the number and title of the manual, and either FAX or mail the photocopy to Alacron.

Remember to include the name and telephone number of the person we should contact if we have questions.

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Telephone: 603-891-2750
FAX: 603-891-2745

Web site:
http://www.alacron.com/

Electronic Mail:
sales@alacron.com
support@alacron.com