

## Using the Universal Signal Processing Library

The Universal Signal Processing Library, or USPL, is an industry standard subroutine library designed to facilitate programming and portability of signal processing code between numerous hardware platforms. Over 240 subroutines make up the current library, with functions that do FFT's, convolutions, filtering, function generation and basic math. The library is based on, and built from the standards set by Floating Point Systems, Inc., the Society of Exploration Geophysicists - Technical Standards Committee, and Quantitative Technology Corporation.

This document includes lots of information regarding the use and operation of the USPL. Other pertinent information is included in several files included with the package:

README.1ST	This contains latest information about the Library and installation notes.
LICENSE.TXT	This is your license agreement, please read it carefully.
REGISTER.TXT	This document describes how to purchase the registered version.

## **New Routines**

Sigma Tech continues to add new routines to the library to round out existing sets of functions, to maintain compatibility with other vendors' scientific subroutine libraries and upon request of customers. Please contact Sigma Tech if you have a requirement not covered by the routines in the USPL.

## Page Format

Each of the USPL routines is described in a format which includes its name, argument declarations from C and FORTRAN, argument and subroutine descriptions, and a mathematical equivalent, or pseudo-code fragment showing the function of the routine. Some routines list restrictions or special conditions, which must be noted for correct results. The format of the pages will allow you to copy and paste code from the routine declaration right into your source code.

## Argument Names

The names of each argument were chosen to provide information about the type of data that they hold. The following table highlights each argument type:

bv	N/A	Bytes
sv	N/A	16-bit Integer
iv	inc, n, ndf, m, fig, idx	32-bit Integer
m	N/A	32-bit Floating Point Matrix
rv	rs	32-bit Floating Point
cv	cs	Complex (32-bit Floating Point pairs)
dm	N/A	64-bit Double Precision FP Matrix
dv	N/A	64-bit Double Precision Floating Point

The first letter describes the data storage format. A "v" as the second letter indicates that the argument is a vector or array. For example, a complex vector would be named "cv". A scalar argument has an "s" for the second letter. Integer scalars sometimes have special names to indicate what they are used for. For example: inc, n, ndf, fig, and idx. Sometimes the names end with a digit to distinguish between multiple vectors of the same type, or to make it clear how to group the arguments. "inc1" is the increment for "rv1" for example. A trailing "o" indicates an output argument; for example "ivo" would be an output integer vector.

## Storage Assumptions

- 1) Bytes and integers are stored in 2's complement format unless they are noted as "unsigned".
- 2) Floating point values are stored according to IEEE-754 format.
- 3) Complex values are stored as two adjacent floating point values, real then imaginary.
- 4) Double precision values are stored as specified by IEEE-754 64-bit floating point format.

## Sample Increments

Most subroutines allow a sample increment to be specified for the vector argument. A sample increment of one specifies the use of every element in the vector. A sample increment of two specifies every other element. An increment of three uses every third element, etc. The very first element stored in the vector is always the first one used, regardless of the sample increment. Element skipping occurs from that point on.

A special case exists for complex vectors. In order to retain compatibility with existing standards, a sample increment of two is used to specify every element of a complex vector. Complex vector increments must always be even numbers. Think of the increment as counting real numbers. (Two reals per complex)

Note that for all increments, the count is by elements, not by bytes. An increment of one for a short integer vector increments by two bytes (one short int), while an increment of one for a real vector increments by four bytes (one real).

## Standard Restrictions

Unless otherwise noted in the description or restrictions section of the each routine, the following notes apply to all USPL routines:

- 1) An output vector may overlap an input vector, provided its sample increment is no greater than that of the input vector. Watch that the input data is not overwritten before it is used.
- 2) All complex vectors must have even sample increments. A sample increment of two indicates a compact complex vector, that is, no elements are skipped.
- 3) If no sample increment argument exists for a vector, that vector is assumed to be compact.
- 4) Data alignment for vectors must correspond to the element size. Complex vectors must start on eight byte boundaries, floating point vectors must start on four byte boundaries, two-byte (short) integer vectors must start on two-byte boundaries, etc.

## Optimization Strategies

The routines have varying levels of optimization depending on the processor/Operating System combination. In general however, several techniques will allow you to get the best performance from your processor/Operating System combination.

- 1) Get the registered version, it has assembly level optimizations for many routines, which feature up to 4 times greater performance.
- 2) Use the routine that has the minimum complexity for the operation you are trying to perform, for example, if you need two multiply two vectors together, use `vmul`, not `vmulsa` as the extra scalar operations may take additional processor cycles if you are not using the scalar. On some platforms such as the i860 which can do a multiply and add per cycle, there is no performance hit since there is an extra add per cycle.
- 3) Using sample increments of one generally increases performance, because some of the routines have logic to eliminate the normal extra multiplies needed for step sizes other than one.
- 4) Minimize the vector size when possible (using blocking techniques) so all operations can happen in L1 or L2 cache (L1 is preferable). The cache sizes vary significantly from processor to processor, so keep this in mind for the target hardware. Also remember in multithreaded operating systems that your process is not the only process running on the CPU, therefore minimizing the vector size will help ensure that your vectors are indeed in the cache.
- 5) Most routines will work with data that is not aligned on 8 byte boundaries, however a huge performance hit is incurred. Always try and align vectors on 8 byte boundaries.
- 6) If possible use the underscore version of the routine ie. `vmul_` as opposed to `vmul`. The non-underscored versions are just a shell that calls the underscore version, so there is a performance hit for pushing/popping values to/from the stack.



## Documentation

Every effort has been made to not only make the documentation easy to find the routine you need, but accurate as well. Due to the large number of routines, it is possible that documentation errors have occurred. If you find the documentation difficult to use or erroneous, please contact Sigma Tech as we are constantly trying to improve all components of the product.

## Technical Support

Please utilize this documentation to assist you in resolving your problem before calling technical support. Resources utilized in technical support reduce our capabilities to enhance the USPL as well as our other products. If you find you still need assistance after utilizing the documentation and trying a routine out in a stand-alone fashion, you can contact us using the information provided in the contact information.

## Contact Information

### Technical Support

email: [support@sigmatechcorp.com](mailto:support@sigmatechcorp.com)

phone: (540) 439-8000

fax: (540) 439-0948

### Product Information

web: [www.sigmatechcorp.com](http://www.sigmatechcorp.com)

email: [uspl@sigmatechcorp.com](mailto:uspl@sigmatechcorp.com)

phone: (540) 439-8000

fax: (540) 439-0948

## Alphabetical Listing of Function Calls

<b><i>Function Name</i></b>	<b><i>Description</i></b>
<a href="#"><u>acorf</u></a>	Frequency Domain Auto Correlation
<a href="#"><u>acort</u></a>	Time Domain Auto Correlation
<a href="#"><u>aspec</u></a>	Accumulating Autospectrum
<a href="#"><u>blkman</u></a>	Apply a Blackman Window to a Real Vector
<a href="#"><u>ccdotp</u></a>	Complex Conjugate Dot Product
<a href="#"><u>ccorf</u></a>	Frequency Domain Cross Correlation
<a href="#"><u>ccort</u></a>	Time Domain Cross Correlation
<a href="#"><u>cdotpr</u></a>	Complex Dot Product
<a href="#"><u>cfft</u></a>	Complex FFT, in Place - Fwd and Inv - Sizes to 1024 Cmplx
<a href="#"><u>cfft</u></a>	Complex Forward FFT - Sizes to 1048576 (1 M) Cmplx
<a href="#"><u>cffti</u></a>	Complex Inverse FFT - Sizes to 1048576 (1 M) Cmplx
<a href="#"><u>fftscl</u></a>	Complex FFT Scale
<a href="#"><u>cft2fc</u></a>	Two Dimensional Complex Forward FFT - Column Compact
<a href="#"><u>cft2fr</u></a>	Two Dimensional Complex Forward FFT - Row Compact
<a href="#"><u>cft2ic</u></a>	Two Dimensional Complex Inverse FFT - Column Compact
<a href="#"><u>cft2ir</u></a>	Two Dimensional Complex Inverse FFT - Row Compact
<a href="#"><u>convd</u></a>	Convolution with Decimation
<a href="#"><u>convrr</u></a>	Convolution Real to Real
<a href="#"><u>convrrs</u></a>	Convolution Real to Real Symmetrical
<a href="#"><u>cpow</u></a>	Complex Vector Power with Add
<a href="#"><u>crvdiv</u></a>	Complex Vector Real Vector Divide
<a href="#"><u>crvmul</u></a>	Complex Vector Real Vector Multiply
<a href="#"><u>cspec</u></a>	Accumulating Cross Spectrum
<a href="#"><u>cvabs</u></a>	Complex Vector Absolute Value
<a href="#"><u>cvadd</u></a>	Complex Vector Add
<a href="#"><u>cvcml</u></a>	Complex Vector Conjugate Multiply
<a href="#"><u>cvcmla</u></a>	Complex Vector Conjugate Multiply with Add
<a href="#"><u>cvcomb</u></a>	Form Complex Vector from Two Real Vectors
<a href="#"><u>cvconj</u></a>	Complex Vector Conjugate
<a href="#"><u>cvcsml</u></a>	Complex Vector Complex Scalar Multiply
<a href="#"><u>cvdiv</u></a>	Complex Vector Complex Vector Division
<a href="#"><u>cvexp</u></a>	Complex Vector Exponential
<a href="#"><u>cvfill</u></a>	Set a Complex Vector to a Complex Scalar
<a href="#"><u>cvma</u></a>	Complex or Conjugate Vector Multiply with Add
<a href="#"><u>cvmags</u></a>	Complex Vector Magnitude Squared
<a href="#"><u>cvmexp</u></a>	Complex Vector Exponential with Multiply
<a href="#"><u>cvmgsa</u></a>	Complex Vector Magnitude Squared with Add
<a href="#"><u>cvml</u></a>	Complex Vector Multiply
<a href="#"><u>cvmla</u></a>	Complex Vector Multiply with Add
<a href="#"><u>cvmov</u></a>	Complex Vector Move
<a href="#"><u>cvmul</u></a>	Complex Vector Multiply or Conjugate Multiply
<a href="#"><u>cvneg</u></a>	Complex Vector Negate
<a href="#"><u>cvphas</u></a>	Complex Vector Negate
<a href="#"><u>cvrcip</u></a>	Complex Vector Reciprocal
<a href="#"><u>cvreal</u></a>	Make a Complex Vector from a Real Vector
<a href="#"><u>cvsmas</u></a>	Scale Complex Vector and Add to Second Complex Vector
<a href="#"><u>cvsmul</u></a>	Complex Vector Scalar Multiply
<a href="#"><u>cvsqrt</u></a>	Complex Vector Square Root
<a href="#"><u>cvsub</u></a>	Complex Vector Subtract
<a href="#"><u>deq22</u></a>	Difference Equation, 2 Poles, 2 Zeroes
<a href="#"><u>dmxmia</u></a>	Double Precision Real Matrix plus Matrix Product
<a href="#"><u>dmxmils</u></a>	Double Precision Real Matrix minus Matrix Product

<a href="#"><u>dmxmov</u></a>	Double Precision Real Matrix Move
<a href="#"><u>dmxmml</u></a>	Double Precision Real Matrix Multiply
<a href="#"><u>dotpr</u></a>	Real Dot Product
<a href="#"><u>envel</u></a>	Vector Envelope
<a href="#"><u>fftwts</u></a>	Create FFT Complex Exponential Tables
<a href="#"><u>fix2n</u></a>	Vector Fix to Two-byte Integer and Round
<a href="#"><u>fix4</u></a>	Vector Fix to Four-byte Integer and Truncate
<a href="#"><u>fix4n</u></a>	Vector Fix to Four-byte Integer and Round
<a href="#"><u>fixbn</u></a>	Vector Fix to One-byte Integer and Round
<a href="#"><u>flt2</u></a>	Float Integer (2 Byte) Vector
<a href="#"><u>flt2iq</u></a>	Float Integer (2 Byte) I,Q pairs and demux 2 channels
<a href="#"><u>flt4</u></a>	Float Integer (4 Byte) Vector
<a href="#"><u>fltb</u></a>	Float Byte
<a href="#"><u>fltbu</u></a>	Float Byte Unsigned
<a href="#"><u>fxsl2n</u></a>	Vector Scale, Limit, & Fix to Two-byte Integer and Round
<a href="#"><u>fxsl4n</u></a>	Vector Scale, Limit, & Fix to Four-byte Integer & Round
<a href="#"><u>fxslbn</u></a>	Vector Scale, Limit, & Fix to One-byte Integer & Round
<a href="#"><u>gcexp</u></a>	Generate Complex Exponential Vector with Constant Rotation
<a href="#"><u>gcosf</u></a>	Vector Generate Cosines
<a href="#"><u>genfilt</u></a>	Generate Complex or Real FIR Filter
<a href="#"><u>genorder</u></a>	Generate Complex or Real FIR Filter Order
<a href="#"><u>hamm</u></a>	Apply a Hamming Window to a Real Vector
<a href="#"><u>hann</u></a>	Apply a Hanning Window to a Real Vector
<a href="#"><u>hlbrt</u></a>	Hilbert Transform
<a href="#"><u>leveldet</u></a>	Time Series Energy Detector
<a href="#"><u>lveq</u></a>	Logical Vector Equal
<a href="#"><u>lvge</u></a>	Logical Vector Greater Than or Equal
<a href="#"><u>lvgt</u></a>	Logical Vector Greater
<a href="#"><u>lvle</u></a>	Logical Vector Less Than or Equal
<a href="#"><u>lvlt</u></a>	Logical Vector Less
<a href="#"><u>lvne</u></a>	Logical Vector Not Equal
<a href="#"><u>lvnot</u></a>	Logical Vector Not Equal to Zero
<a href="#"><u>maxmgv</u></a>	Maximum Magnitude Element of Vector
<a href="#"><u>maxv</u></a>	Find the Maximum Value and its Location
<a href="#"><u>meamgv</u></a>	Mean Magnitude of Real Vector
<a href="#"><u>meanv</u></a>	Mean of Real Vector
<a href="#"><u>measqv</u></a>	Mean of Square of Real Vector
<a href="#"><u>minmgv</u></a>	Find the Minimum Magnitude and its Location
<a href="#"><u>minv</u></a>	Find the Minimum Value and its Position
<a href="#"><u>mvessq</u></a>	Mean of Signed Square of Vector Elements
<a href="#"><u>mxmla</u></a>	Real Matrix plus Matrix Product
<a href="#"><u>mxmls</u></a>	Real Matrix minus Matrix Product
<a href="#"><u>mxmov</u></a>	Real Matrix Move
<a href="#"><u>mxmml</u></a>	Real Matrix Multiply
<a href="#"><u>polar</u></a>	Rectangular to Polar Coordinate Conversion
<a href="#"><u>rect</u></a>	Polar to Rectangular Coordinate Conversion
<a href="#"><u>reqs</u></a>	Find Location of First Element Equal to a Scalar
<a href="#"><u>rfft</u></a>	Real FFT, in Place - Fwd and Inv - Sizes to 2048 Reals
<a href="#"><u>rfft</u></a>	Real Forward FFT - Sizes to 2097152 (2 Meg) Reals
<a href="#"><u>rfti</u></a>	Real Inverse FFT - Sizes to 2097152 (2 Meg) Reals
<a href="#"><u>rftsc</u></a>	Real FFT Scale and Format
<a href="#"><u>rft2fc</u></a>	Two Dimensional Real Forward FFT - Column Compact
<a href="#"><u>rft2fr</u></a>	Two Dimensional Real Forward FFT - Row Compact
<a href="#"><u>rft2ic</u></a>	Two Dimensional Real Inverse FFT - Column Compact
<a href="#"><u>rft2ir</u></a>	Two Dimensional Real Inverse FFT - Row Compact
<a href="#"><u>rges</u></a>	Find Location of First Element Greater/Equal to a Scalar

<a href="#"><u>rgts</u></a>	Find Location of First Element Greater Than a Scalar
<a href="#"><u>rlts</u></a>	Find Location of First Element Less Than a Scalar
<a href="#"><u>rmax</u></a>	Find the Maximum Value and its Location
<a href="#"><u>rmaxmg</u></a>	Find the Maximum Magnitude
<a href="#"><u>rmin</u></a>	Find the Minimum Value and its Position
<a href="#"><u>rminmg</u></a>	Find the Minimum Magnitude
<a href="#"><u>rmsqv</u></a>	Root Mean Square of a Real Vector
<a href="#"><u>rnes</u></a>	Find Position of First Element Not Equal to a Scalar
<a href="#"><u>rsve</u></a>	Running Sum of Real Vector
<a href="#"><u>shphu</u></a>	Schafer's Phase Unwrapping
<a href="#"><u>shphuf</u></a>	Schafer's Phase Unwrapping, Fraction of a Circle Argument
<a href="#"><u>sn2</u></a>	Sum the Squared Difference Between Two Vectors
<a href="#"><u>sdiv</u></a>	Divide Scalar by Vector
<a href="#"><u>sve</u></a>	Sum of Real Vector
<a href="#"><u>svemg</u></a>	Sum of Vector Magnitudes
<a href="#"><u>svesq</u></a>	Sum of Vector Elements Squared
<a href="#"><u>svessq</u></a>	Sum of Vector Element Signed Squares
<a href="#"><u>tconv</u></a>	Tapered Convolution
<a href="#"><u>trans</u></a>	Complex Vector Divided by Real Vector (Transfer)
<a href="#"><u>vaam</u></a>	Vector Add, Add, and Multiply
<a href="#"><u>vabs</u></a>	Vector Absolute Value
<a href="#"><u>vacos</u></a>	Vector Arccosine
<a href="#"><u>vadd</u></a>	Add Two Vectors
<a href="#"><u>vaint</u></a>	Vector Align to Integer
<a href="#"><u>vam</u></a>	Vector Add and Multiply
<a href="#"><u>vanint</u></a>	Vector Align to Nearest Integer
<a href="#"><u>vasbm</u></a>	Vector Add, Subtract, and Multiply
<a href="#"><u>vasin</u></a>	Vector Arcsine
<a href="#"><u>vasm</u></a>	Vector Add and Scalar Multiply
<a href="#"><u>vatan</u></a>	Vector Arctangent
<a href="#"><u>vatan2</u></a>	Vector Arctangent of Two Arguments
<a href="#"><u>vatan2f</u></a>	Vector Arctangent of Two Arguments in Fractions
<a href="#"><u>vavexp</u></a>	Vector Exponential Averaging
<a href="#"><u>vavlin</u></a>	Vector Linear Averaging
<a href="#"><u>vclip</u></a>	Vector Clip
<a href="#"><u>vclr</u></a>	Zero a Vector
<a href="#"><u>vcmprs</u></a>	Vector Compress
<a href="#"><u>vcos</u></a>	Vector Cosine
<a href="#"><u>vcosf</u></a>	Vector Cosine in Fractions of a Circle
<a href="#"><u>vdbpwr</u></a>	Vector Conversion to dB
<a href="#"><u>vdiv</u></a>	Divide One Vector by Another
<a href="#"><u>vdpsp</u></a>	Vector Convert Double to Single Precision
<a href="#"><u>veud2</u></a>	Vector Euclidean Distance
<a href="#"><u>veud3</u></a>	Vector Euclidean Distance (3 Dimensional)
<a href="#"><u>vexp</u></a>	Vector Exponentiation
<a href="#"><u>vexp10</u></a>	Vector Base 10 Exponential
<a href="#"><u>vexp2</u></a>	Vector Base 2 Exponential
<a href="#"><u>vfill</u></a>	Set a Vector in Memory to a Scalar Value
<a href="#"><u>vfrac</u></a>	Vector Truncate to Fraction
<a href="#"><u>vfracn</u></a>	Vector Truncate to Nearest Fraction
<a href="#"><u>vgathr</u></a>	Vector Gather
<a href="#"><u>vgen</u></a>	Generate a Vector in Memory
<a href="#"><u>viadd</u></a>	Add Two Integer Vectors
<a href="#"><u>viand</u></a>	And Two Integer Vectors
<a href="#"><u>viars</u></a>	Integer Vector Arithmetic Right Shift
<a href="#"><u>viclip</u></a>	Vector Inverse Clip

<a href="#"><u>viis</u></a>	Integer Vector Left Shift
<a href="#"><u>vimag</u></a>	Extract Imaginary Part of Complex Vector
<a href="#"><u>vimul</u></a>	Add Two Integer Vectors
<a href="#"><u>vindex</u></a>	Vector Index, Truncate
<a href="#"><u>vineg</u></a>	Integer Vector Negate
<a href="#"><u>vinth</u></a>	Vector Interpolate
<a href="#"><u>vior</u></a>	OR Two Integer Vectors
<a href="#"><u>virs</u></a>	Integer Vector Right Shift
<a href="#"><u>visub</u></a>	Subtract Two Integer Vectors
<a href="#"><u>vixor</u></a>	Exclusive OR (XOR) Two Integer Vectors
<a href="#"><u>vlim</u></a>	Vector Limit
<a href="#"><u>vlint</u></a>	Vector Linear Interpolate
<a href="#"><u>vlmerg</u></a>	Vector Logical Merge
<a href="#"><u>vlog</u></a>	Natural Logarithm of a Vector
<a href="#"><u>vlog10</u></a>	Vector Base 10 Logarithm
<a href="#"><u>vlog2</u></a>	Vector Base 2 Logarithm
<a href="#"><u>vma</u></a>	Vector Multiply and Add
<a href="#"><u>vmax</u></a>	Vector Maximum
<a href="#"><u>vmaxmg</u></a>	Vector Maximum Magnitude
<a href="#"><u>vmin</u></a>	Vector Minimum
<a href="#"><u>vminmg</u></a>	Vector Minimum Magnitude
<a href="#"><u>mma</u></a>	Vector Multiply, Multiply, and Add
<a href="#"><u>mmmsb</u></a>	Vector Multiply, Multiply, and Subtract
<a href="#"><u>mov</u></a>	Copy One Vector to Another
<a href="#"><u>vmsa</u></a>	Vector Multiply and Scalar Add
<a href="#"><u>vmsb</u></a>	Vector Multiply and Subtract
<a href="#"><u>vmul</u></a>	Multiply Two Vectors
<a href="#"><u>vnabs</u></a>	Vector Negative Absolute Value
<a href="#"><u>vneg</u></a>	Negate a Vector
<a href="#"><u>vnmsa</u></a>	Vector Negative Multiply and Scalar Add
<a href="#"><u>vpmerg</u></a>	Vector Positive Merge
<a href="#"><u>vpoly</u></a>	Vector Polynomial Evaluation
<a href="#"><u>vqint</u></a>	Vector Quadratic Interpolate
<a href="#"><u>vramp</u></a>	Generate a Ramp in a Vector
<a href="#"><u>vrand</u></a>	Generate Random Number Vector
<a href="#"><u>vreal</u></a>	Extract Real Part of Complex Vector
<a href="#"><u>vrecip</u></a>	Reciprocal of Vector
<a href="#"><u>vrsgt</u></a>	Vector Reciprocal Square Root
<a href="#"><u>vrvs</u></a>	Reverse a Vector
<a href="#"><u>vsadd</u></a>	Add a Scalar to a Vector
<a href="#"><u>vsbm</u></a>	Vector Subtract and Multiply
<a href="#"><u>vsbsbm</u></a>	Vector Subtract, Subtract, and Multiply
<a href="#"><u>vsbsm</u></a>	Vector Subtract and Scalar Multiply
<a href="#"><u>vscatr</u></a>	Vector Scatter
<a href="#"><u>vsdiv</u></a>	Divide Vector by Scalar
<a href="#"><u>vsimps</u></a>	Simpson's Rule Integration
<a href="#"><u>vsin</u></a>	Vector Sine
<a href="#"><u>vsinf</u></a>	Vector Sine in Fractions
<a href="#"><u>vsinrf</u></a>	Vector Sine in Fractions, Reduced Range
<a href="#"><u>vsm2sa</u></a>	Multiply Two Vectors by Scalars and Add a Scalar
<a href="#"><u>vma</u></a>	Vector Scalar Multiply and Add
<a href="#"><u>vma2</u></a>	Two Vector Multiply and Scalar Add
<a href="#"><u>vma3</u></a>	Three Vector Multiply and Scalar Add
<a href="#"><u>vma4</u></a>	Four Vector Multiply and Scalar Add
<a href="#"><u>vmsa</u></a>	Multiply Vector by a Scalar and Add Scalar
<a href="#"><u>vmsb</u></a>	Vector Scalar Multiply and Subtract

<a href="#"><u>vsmul</u></a>	Multiply Vector by a Scalar
<a href="#"><u>vspdp</u></a>	Vector Convert Single to Double Precision
<a href="#"><u>vsq</u></a>	Vector Square
<a href="#"><u>vsqrt</u></a>	Vector Square Root
<a href="#"><u>vssg</u></a>	Vector Signed Square
<a href="#"><u>vsub</u></a>	Subtract One Vector from Another
<a href="#"><u>vsum</u></a>	Vector Sum
<a href="#"><u>vswap</u></a>	Vector Swap
<a href="#"><u>vtabi</u></a>	Vector Table Look-up, Linear Interpolate
<a href="#"><u>vtan</u></a>	Vector Tangent
<a href="#"><u>vtanf</u></a>	Vector Tangent in Fractions of a Circle
<a href="#"><u>vthr</u></a>	Vector Threshold
<a href="#"><u>vthres</u></a>	Replace Elements Less than Scalar with Zero
<a href="#"><u>vtrapz</u></a>	Trapezoidal Rule Integration
<a href="#"><u>vxcs</u></a>	Real Vector Multiplied by Complex Exponential
<a href="#"><u>vxcsf</u></a>	Real Vector Multiplied by Fractional Complex Exponential
<a href="#"><u>wiener</u></a>	Wiener-Levinson Equation Solution

## Basic Vector Arithmetic

<a href="#"><u>vclr</u></a>	Zero a Vector
<a href="#"><u>vfill</u></a>	Set a Vector in Memory to a Scalar Value
<a href="#"><u>vramp</u></a>	Generate a Ramp in a Vector
<a href="#"><u>vgen</u></a>	Generate a Vector in Memory
<a href="#"><u>gcosf</u></a>	Vector Generate Cosines
<a href="#"><u>gcexp</u></a>	Generate Complex Exponential Vector with Constant Rotation
<a href="#"><u>vmov</u></a>	Copy One Vector to Another
<a href="#"><u>vswap</u></a>	Vector Swap
<a href="#"><u>vrvs</u></a>	Reverse a Vector
<a href="#"><u>vneg</u></a>	Negate a Vector
<a href="#"><u>vrecip</u></a>	Reciprocal of Vector
<a href="#"><u>vsadd</u></a>	Add a Scalar to a Vector
<a href="#"><u>vsdiv</u></a>	Divide Vector by Scalar
<a href="#"><u>sdiv</u></a>	Divide Scalar by Vector
<a href="#"><u>vsmul</u></a>	Multiply Vector by a Scalar
<a href="#"><u>vsmsa</u></a>	Multiply Vector by a Scalar and Add Scalar
<a href="#"><u>vsm2sa</u></a>	Multiply Two Vectors by Scalars and Add a Scalar
<a href="#"><u>vadd</u></a>	Add Two Vectors
<a href="#"><u>vsub</u></a>	Subtract One Vector from Another
<a href="#"><u>vmul</u></a>	Multiply Two Vectors
<a href="#"><u>vdiv</u></a>	Divide One Vector by Another
<a href="#"><u>vsma</u></a>	Vector Scalar Multiply and Add
<a href="#"><u>vsmsb</u></a>	Vector Scalar Multiply and Subtract
<a href="#"><u>vasm</u></a>	Vector Add and Scalar Multiply
<a href="#"><u>vsbsm</u></a>	Vector Subtract and Scalar Multiply
<a href="#"><u>vintb</u></a>	Vector Interpolate
<a href="#"><u>vmsa</u></a>	Vector Multiply and Scalar Add
<a href="#"><u>vnmsa</u></a>	Vector Negative Multiply and Scalar Add
<a href="#"><u>vsma2</u></a>	Two Vector Multiply and Scalar Add
<a href="#"><u>vsma3</u></a>	Three Vector Multiply and Scalar Add
<a href="#"><u>vsma4</u></a>	Four Vector Multiply and Scalar Add
<a href="#"><u>vma</u></a>	Vector Multiply and Add
<a href="#"><u>vmsb</u></a>	Vector Multiply and Subtract



<a href="#"><u>vam</u></a>	Vector Add and Multiply
<a href="#"><u>vsbm</u></a>	Vector Subtract and Multiply
<a href="#"><u>vaam</u></a>	Vector Add, Add, and Multiply
<a href="#"><u>vasbm</u></a>	Vector Add, Subtract, and Multiply
<a href="#"><u>vsbsbm</u></a>	Vector Subtract, Subtract, and Multiply
<a href="#"><u>vmma</u></a>	Vector Multiply, Multiply, and Add
<a href="#"><u>vmmsb</u></a>	Vector Multiply, Multiply, and Subtract

## Non-Linear Vector Arithmetic; Intrinsic

<a href="#"><u>vabs</u></a>	Vector Absolute Value
<a href="#"><u>vnabs</u></a>	Vector Negative Absolute Value
<a href="#"><u>vsq</u></a>	Vector Square
<a href="#"><u>vssq</u></a>	Vector Signed Square
<a href="#"><u>vsqrt</u></a>	Vector Square Root
<a href="#"><u>vrsqrt</u></a>	Vector Reciprocal Square Root
<a href="#"><u>veud2</u></a>	Vector Euclidean Distance
<a href="#"><u>veud3</u></a>	Vector Euclidean Distance (3 Dimensional)
<a href="#"><u>vaint</u></a>	Vector Align to Integer
<a href="#"><u>vanint</u></a>	Vector Align to Nearest Integer
<a href="#"><u>vfrac</u></a>	Vector Truncate to Fraction
<a href="#"><u>vfracn</u></a>	Vector Truncate to Nearest Fraction
<a href="#"><u>vcos</u></a>	Vector Cosine
<a href="#"><u>vcosf</u></a>	Vector Cosine in Fractions of a Circle
<a href="#"><u>vsin</u></a>	Vector Sine
<a href="#"><u>vsinf</u></a>	Vector Sine in Fractions
<a href="#"><u>vsinrf</u></a>	Vector Sine in Fractions, Reduced Range
<a href="#"><u>vtan</u></a>	Vector Tangent
<a href="#"><u>vtanf</u></a>	Vector Tangent in Fractions of a Circle
<a href="#"><u>vacos</u></a>	Vector Arccosine
<a href="#"><u>vasin</u></a>	Vector Arcsine
<a href="#"><u>vatan</u></a>	Vector Arctangent
<a href="#"><u>vatan2</u></a>	Vector Arctangent of Two Arguments
<a href="#"><u>vatan2f</u></a>	Vector Arctangent of Two Arguments in Fractions
<a href="#"><u>vexp</u></a>	Vector Exponentiation
<a href="#"><u>vexp10</u></a>	Vector Base 10 Exponential
<a href="#"><u>vexp2</u></a>	Vector Base 2 Exponential
<a href="#"><u>vlog</u></a>	Natural Logarithm of a Vector
<a href="#"><u>vlog10</u></a>	Vector Base 10 Logarithm
<a href="#"><u>vlog2</u></a>	Vector Base 2 Logarithm
<a href="#"><u>vdbpwr</u></a>	Vector Conversion to dB
<a href="#"><u>vthr</u></a>	Vector Threshold
<a href="#"><u>vtres</u></a>	Replace Elements Less than Scalar with Zero
<a href="#"><u>vmax</u></a>	Vector Maximum
<a href="#"><u>vmaxmg</u></a>	Vector Maximum Magnitude
<a href="#"><u>vmin</u></a>	Vector Minimum
<a href="#"><u>vminmg</u></a>	Vector Minimum Magnitude
<a href="#"><u>vlim</u></a>	Vector Limit
<a href="#"><u>vclip</u></a>	Vector Clip
<a href="#"><u>viclip</u></a>	Vector Inverse Clip
<a href="#"><u>lveq</u></a>	Logical Vector Equal
<a href="#"><u>lvge</u></a>	Logical Vector Greater Than or Equal
<a href="#"><u>lvgt</u></a>	Logical Vector Greater
<a href="#"><u>lvle</u></a>	Logical Vector Less Than or Equal
<a href="#"><u>lvlt</u></a>	Logical Vector Less
<a href="#"><u>lvne</u></a>	Logical Vector Not Equal
<a href="#"><u>lvnot</u></a>	Logical Vector Not Equal to Zero



## Complex Vector Routines

<a href="#"><u>cvfill</u></a>	Set a Complex Vector to a Complex Scalar
<a href="#"><u>cvmov</u></a>	Complex Vector Move
<a href="#"><u>cvneg</u></a>	Complex Vector Negate
<a href="#"><u>cvconj</u></a>	Complex Vector Conjugate
<a href="#"><u>cvabs</u></a>	Complex Vector Absolute Value
<a href="#"><u>cvphas</u></a>	Complex Vector Negate
<a href="#"><u>cvmags</u></a>	Complex Vector Magnitude Squared
<a href="#"><u>cvrcip</u></a>	Complex Vector Reciprocal
<a href="#"><u>cvsqrt</u></a>	Complex Vector Square Root
<a href="#"><u>cvsmul</u></a>	Complex Vector Scalar Multiply
<a href="#"><u>cvcsml</u></a>	Complex Vector Complex Scalar Multiply
<a href="#"><u>cvsmc</u></a>	Scale Complex Vector and Add to Second Complex Vector
<a href="#"><u>cvadd</u></a>	Complex Vector Add
<a href="#"><u>cvsub</u></a>	Complex Vector Subtract
<a href="#"><u>cvml</u></a>	Complex Vector Multiply
<a href="#"><u>cvcmml</u></a>	Complex Vector Conjugate Multiply
<a href="#"><u>cvmul</u></a>	Complex Vector Multiply or Conjugate Multiply
<a href="#"><u>cvdiv</u></a>	Complex Vector Complex Vector Division
<a href="#"><u>crvmul</u></a>	Complex Vector Real Vector Multiply
<a href="#"><u>crvdiv</u></a>	Complex Vector Real Vector Divide
<a href="#"><u>trans</u></a>	Complex Vector Divided by Real Vector (Transfer)
<a href="#"><u>aspec</u></a>	Accumulating Autospectrum
<a href="#"><u>cvmgsc</u></a>	Complex Vector Magnitude Squared with Add
<a href="#"><u>cpow</u></a>	Complex Vector Power with Add
<a href="#"><u>cvmla</u></a>	Complex Vector Multiply with Add
<a href="#"><u>cvcmmla</u></a>	Complex Vector Conjugate Multiply with Add
<a href="#"><u>cvma</u></a>	Complex or Conjugate Vector Multiply with Add
<a href="#"><u>cspec</u></a>	Accumulating Cross Spectrum
<a href="#"><u>cvexp</u></a>	Complex Vector Exponential
<a href="#"><u>cvmexp</u></a>	Complex Vector Exponential with Multiply
<a href="#"><u>vxcs</u></a>	Real Vector Multiplied by Complex Exponential
<a href="#"><u>vxcsf</u></a>	Real Vector Multiplied by Fractional Complex Exponential
<a href="#"><u>rect</u></a>	Polar to Rectangular Coordinate Conversion
<a href="#"><u>polar</u></a>	Rectangular to Polar Coordinate Conversion

## Integer Vector Arithmetic

<a href="#"><u>vineg</u></a>	Integer Vector Negate
<a href="#"><u>viadd</u></a>	Add Two Integer Vectors
<a href="#"><u>visub</u></a>	Subtract Two Integer Vectors
<a href="#"><u>vimul</u></a>	Add Two Integer Vectors
<a href="#"><u>viis</u></a>	Integer Vector Left Shift
<a href="#"><u>virs</u></a>	Integer Vector Right Shift
<a href="#"><u>viars</u></a>	Integer Vector Arithmetic Right Shift
<a href="#"><u>vior</u></a>	OR Two Integer Vectors
<a href="#"><u>viand</u></a>	And Two Integer Vectors
<a href="#"><u>vixor</u></a>	Exclusive OR (XOR) Two Integer Vectors

## Matrix Functions

[dmxm1a](#)

Double Precision Real Matrix plus Matrix Product

[dmxm1s](#)

Double Precision Real Matrix minus Matrix Product

[dmxm1ov](#)

Double Precision Real Matrix Move

[dmxm1ul](#)

Double Precision Real Matrix Multiply

[mxm1a](#)

Real Matrix plus Matrix Product

[mxm1s](#)

Real Matrix minus Matrix Product

[mxm1ov](#)

Real Matrix Move

[mxm1ul](#)

Real Matrix Multiply

[wiener](#)

Wiener-Levinson Equation Solution

## Vector Reduction Routines

<a href="#"><u>dotpr</u></a>	Real Dot Product
<a href="#"><u>cdotpr</u></a>	Complex Dot Product
<a href="#"><u>ccdotp</u></a>	Complex Conjugate Dot Product
<a href="#"><u>meanv</u></a>	Mean of Real Vector
<a href="#"><u>meamgv</u></a>	Mean Magnitude of Real Vector
<a href="#"><u>measqv</u></a>	Mean of Square of Real Vector
<a href="#"><u>mvessq</u></a>	Mean of Signed Square of Vector Elements
<a href="#"><u>rmsqv</u></a>	Root Mean Square of a Real Vector
<a href="#"><u>sve</u></a>	Sum of Real Vector
<a href="#"><u>rsve</u></a>	Running Sum of Real Vector
<a href="#"><u>svermg</u></a>	Sum of Vector Magnitudes
<a href="#"><u>svesq</u></a>	Sum of Vector Elements Squared
<a href="#"><u>svessq</u></a>	Sum of Vector Element Signed Squares
<a href="#"><u>sn2</u></a>	Sum the Squared Difference Between Two Vectors
<a href="#"><u>maxmgv</u></a>	Maximum Magnitude Element of Vector
<a href="#"><u>maxv</u></a>	Find the Maximum Value and its Location
<a href="#"><u>minmgv</u></a>	Find the Minimum Magnitude and its Location
<a href="#"><u>minv</u></a>	Find the Minimum Value and its Position
<a href="#"><u>rmax</u></a>	Find the Maximum Value and its Location
<a href="#"><u>rmin</u></a>	Find the Minimum Value and its Position
<a href="#"><u>rmaxmg</u></a>	Find the Maximum Magnitude
<a href="#"><u>rminmg</u></a>	Find the Minimum Magnitude
<a href="#"><u>rges</u></a>	Find Location of First Element Greater/Equal to a Scalar
<a href="#"><u>regs</u></a>	Find Location of First Element Equal to a Scalar
<a href="#"><u>rnes</u></a>	Find Position of First Element Not Equal to a Scalar
<a href="#"><u>rlts</u></a>	Find Location of First Element Less Than a Scalar

## Format Conversion Routines

<a href="#"><u>vreal</u></a>	Extract Real Part of Complex Vector
<a href="#"><u>vimag</u></a>	Extract Imaginary Part of Complex Vector
<a href="#"><u>cvcomb</u></a>	Form Complex Vector from Two Real Vectors
<a href="#"><u>creal</u></a>	Make a Complex Vector from a Real Vector
<a href="#"><u>vspdp</u></a>	Vector Convert Single to Double Precision
<a href="#"><u>vdpsp</u></a>	Vector Convert Double to Single Precision
<a href="#"><u>flt2</u></a>	Float Integer (2 Byte) Vector
<a href="#"><u>flt2iq</u></a>	Float Integer (2 Byte) I,Q pairs and demux 2 channels
<a href="#"><u>flt4</u></a>	Float Integer (4 Byte) Vector
<a href="#"><u>fltb</u></a>	Float Byte
<a href="#"><u>fltbu</u></a>	Float Byte Unsigned
<a href="#"><u>fix2n</u></a>	Vector Fix to Two-byte Integer and Round
<a href="#"><u>fix4</u></a>	Vector Fix to Four-byte Integer and Truncate
<a href="#"><u>fix4n</u></a>	Vector Fix to Four-byte Integer and Round
<a href="#"><u>fixbn</u></a>	Vector Fix to One-byte Integer and Round
<a href="#"><u>fxsl2n</u></a>	Vector Scale, Limit, & Fix to Two-byte Integer and Round
<a href="#"><u>fxsl4n</u></a>	Vector Scale, Limit, & Fix to Four-byte Integer & Round
<a href="#"><u>fxslbn</u></a>	Vector Scale, Limit, & Fix to One-byte Integer & Round



## Integration and Filtering Routines

<a href="#"><u>vsum</u></a>	Vector Sum
<a href="#"><u>deq22</u></a>	Difference Equation, 2 Poles, 2 Zeroes
<a href="#"><u>vsimps</u></a>	Simpson's Rule Integration
<a href="#"><u>vtrapz</u></a>	Trapezoidal Rule Integration
<a href="#"><u>convd</u></a>	Convolution with Decimation
<a href="#"><u>convrr</u></a>	Convolution Real to Real
<a href="#"><u>convrrs</u></a>	Convolution Real to Real Symmetrical
<a href="#"><u>tconv</u></a>	Tapered Convolution
<a href="#"><u>genorder</u></a>	Generate Complex or Real FIR Filter Order
<a href="#"><u>genfilt</u></a>	Generate Complex or Real FIR Filter
<a href="#"><u>acort</u></a>	Time Domain Auto Correlation
<a href="#"><u>ccort</u></a>	Time Domain Cross Correlation
<a href="#"><u>acorf</u></a>	Frequency Domain Auto Correlation
<a href="#"><u>ccorf</u></a>	Frequency Domain Cross Correlation
<a href="#"><u>vavexp</u></a>	Vector Exponential Averaging
<a href="#"><u>vavlin</u></a>	Vector Linear Averaging

## Fast Fourier Transform (FFT) Routines

<a href="#"><u>cfft</u></a>	Complex Forward FFT - Sizes to 1048576 (1 M) Cmplx
<a href="#"><u>cfti</u></a>	Complex Inverse FFT - Sizes to 1048576 (1 M) Cmplx
<a href="#"><u>cfft</u></a>	Complex FFT, in Place - Fwd and Inv - Sizes to 1024 Cmplx
<a href="#"><u>cfftsc</u></a>	Complex FFT Scale
<a href="#"><u>cft2fc</u></a>	Two Dimensional Complex Forward FFT - Column Compact
<a href="#"><u>cft2fr</u></a>	Two Dimensional Complex Forward FFT - Row Compact
<a href="#"><u>cft2ic</u></a>	Two Dimensional Complex Inverse FFT - Column Compact
<a href="#"><u>cft2ir</u></a>	Two Dimensional Complex Inverse FFT - Row Compact
<a href="#"><u>rfft</u></a>	Real Forward FFT - Sizes to 2097152 (2 Meg) Reals
<a href="#"><u>rfti</u></a>	Real Inverse FFT - Sizes to 2097152 (2 Meg) Reals
<a href="#"><u>rfft</u></a>	Real FFT, in Place - Fwd and Inv - Sizes to 2048 Reals
<a href="#"><u>rfftsc</u></a>	Real FFT Scale and Format
<a href="#"><u>rft2fc</u></a>	Two Dimensional Real Forward FFT - Column Compact
<a href="#"><u>rft2fr</u></a>	Two Dimensional Real Forward FFT - Row Compact
<a href="#"><u>rft2ic</u></a>	Two Dimensional Real Inverse FFT - Column Compact
<a href="#"><u>rft2ir</u></a>	Two Dimensional Real Inverse FFT - Row Compact
<a href="#"><u>fftwts</u></a>	Create FFT Complex Exponential Tables

## Miscellaneous Real Vector Routines

<a href="#"><u>vpoly</u></a>	Vector Polynomial Evaluation
<a href="#"><u>vrand</u></a>	Generate Random Number Vector
<a href="#"><u>vcmprrs</u></a>	Vector Compress
<a href="#"><u>vindex</u></a>	Vector Index, Truncate
<a href="#"><u>vgathr</u></a>	Vector Gather
<a href="#"><u>vscatr</u></a>	Vector Scatter
<a href="#"><u>vlint</u></a>	Vector Linear Interpolate
<a href="#"><u>vqint</u></a>	Vector Quadratic Interpolate
<a href="#"><u>vtabi</u></a>	Vector Table Look-up, Linear Interpolate
<a href="#"><u>shphu</u></a>	Schafer's Phase Unwrapping
<a href="#"><u>shphuf</u></a>	Schafer's Phase Unwrapping, Fraction of a Circle Argument
<a href="#"><u>hlbrt</u></a>	Hilbert Transform
<a href="#"><u>envel</u></a>	Vector Envelope
<a href="#"><u>vlmerg</u></a>	Vector Logical Merge
<a href="#"><u>vpmerg</u></a>	Vector Positive Merge
<a href="#"><u>blkman</u></a>	Apply a Blackman Window to a Real Vector
<a href="#"><u>hamm</u></a>	Apply a Hamming Window to a Real Vector
<a href="#"><u>hann</u></a>	Apply a Hanning Window to a Real Vector
<a href="#"><u>leveldet</u></a>	Time Series Energy Detector

## **acorf**

## *Frequency Domain Auto Correlation*

### **C USAGE:**

```
float rv1[], rvo[];
long no, n1;
acorf_(rv1, rvo, &no, &n1);
-or-
acorf(rv1, rvo, no, n1);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 no, n1
call acorf(rv1, rvo, no, n1)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(i/o) - Real input vector
rvo	(out) - Real output vector
no	(inp) - Number of points in output vector
n1	(inp) - Number of points in input vector

### **SUBROUTINE DESCRIPTION:**

Auto correlates in the frequency domain the input vector, rv1, with a size of n1, producing the output vector, rvo, with a size no.

### **MATHEMATICAL EQUIVALENT:**

$$rvo(m+inco) = \text{SUM} \{ rv1[m+j-1] * rv1[j], j=1 \text{ to } n1-m+1 \}, \text{ for } m=1 \text{ to } no$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) The storage allocated by rv1 must be twice the size of n1.
- 2) rv1 will be overwritten.
- 3) n1 must be a power of 2,  $8 \leq n1 \leq 1024$ .
- 4) All vectors must be compact.

## **acort**

## *Time Domain Auto Correlation*

### **C USAGE:**

```
float rv1[], rvo[];
long no, n1;
acort_(rv1, rvo, &no, &n1);
-or-
acort(rv1, rvo, no, n1);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 no, n1
call acort(rv1, rvo, no, n1)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
rvo	(out) - Real output vector
no	(inp) - Number of points in output vector
n1	(inp) - Number of points of input vector

### **SUBROUTINE DESCRIPTION:**

Auto correlates in the time domain the input vector, rv1, with a size of n1, producing the output vector, rvo, with a size of no.

### **MATHEMATICAL EQUIVALENT:**

$$rvo(inco) = \text{SUM} \{ rv1[m+j-1] * rv1[j], j=1 \text{ to } n1-m+1 \}, \text{ for } m=1 \text{ to } no$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

All vectors must be compact.

## **aspec**

## *Accumulating Autospectrum*

### **C USAGE:**

```
float cv1[][2], rvio[];  
long incl, incio, n;  
aspec_(cv1, &incl, rvio, &incio, &n);  
-or-  
aspec(cv1, incl, rvio, incio, n);
```

### **FORTRAN USAGE:**

```
complex cv1()  
real*4 rvio()  
integer*4 incl, incio, n  
call aspec(cv1, incl, rvio, incio, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Complex vector increment
rvio	(i/o) - Real input/output vector
incio	(inp) - Real vector increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Each element of the complex input vector, cv1, is multiplied by its conjugate and added to the real input vector, rvio. The result is placed back in the real vector, rvio, becoming the real output vector. (This routine is included only for backwards compatibility with other subroutine libraries. For new programs, use cvmgsa.)

### **MATHEMATICAL EQUIVALENT:**

$$rvio[i*incio] = cv1[i*incl] * \text{CONJ} \{cv1[i*incl]\} + rvio[i*incio], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **blkman**

*Apply a Blackman Window to a Real Vector*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
blkman_(rv1, &incl, rvo, &inco, &n);
-or-
blkman(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call blkman(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Multiplies each element of the input vector rv1 by the Blackman window.

### **MATHEMATICAL EQUIVALENT:**

for j=0 to n-1:  
$$rvo[j*inco] = rv1[j*incl] * (0.42 - 0.5 * \cos\{2\pi*j/n\} + 0.08 * \cos\{4\pi*j/n\})$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **ccdotp**

## *Complex Conjugate Dot Product*

### **C USAGE:**

```
float cv1[][2], cv2[][2], cso[2];
long inc1, inc2, n;
ccdotp_(cv1, &inc1, cv2, &inc2, cso, &n);
-or-
ccdotp(cv1, inc1, cv2, inc2, cso, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cv2(), cso
integer*4 inc1, inc2, n
call ccdotp(cv1, inc1, cv2, inc2, cso, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector 1
inc1	(inp) - Vector increment for cv1
cv2	(inp) - Complex input vector 2
inc2	(inp) - Vector increment for cv2
cso	(out) - Complex scalar output
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Second complex vector is multiplied by the conjugate of the first and summed into scalar cso.

### **MATHEMATICAL EQUIVALENT:**

$$cso = \text{SUM} \{ \text{CONJ} \{ cv1[i*inc1] \} * cv2[i*inc2] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **ccorf**

## *Frequency Domain Cross Correlation*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long no, n1;
ccorf-(rv1, rv2, rvo, &no, &n1);
-or-
ccorf(rv1, rv2, rvo, no, n1);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 no, n1
call ccorf(rv1, rv2, rvo, no, n1)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
rv2	(inp) - Second real input vector
rvo	(out) - Real output vector
no	(inp) - Number of points in output vector
n1	(inp) - Number of points of vectors rv1 and rv2

### **SUBROUTINE DESCRIPTION:**

Cross correlates, in the frequency domain, the inputs vectors rv1 and rv2 of size n1, and produces the output vector, rvo, with a size of no.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[inco] = \text{SUM} \{ rv1[m+j-1] * rv2[j], j=1 \text{ to } n1-m+1 \}, \text{ for } m = 1 \text{ to } no$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) The size of rv1 and rv2 must be at least  $2 * n1$ .
- 2) n1 must be a power of 2,  $8 \leq n \leq 1024$ .
- 3) All vectors must be compact.

## **ccort**

## *Time Domain Cross Correlation*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long no, n1;
ccort_(rv1, rv2, rvo, &no, &n1);
-or-
ccort(rv1, rv2, rvo, no, n1);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 no, n1
call ccort(rv1, rv2, rvo, no, n1)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
rv2	(inp) - Second real input vector
rvo	(out) - Real output vector
no	(inp) - Number of points in output vector
n1	(inp) - Number of points in vectors rv1 and rv2

### **SUBROUTINE DESCRIPTION:**

Cross correlates, in the time domain, the input vectors rv1 and rv2, with a size of n1, and produces the output vector, rvo, with a size of no.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[inco] = \text{SUM} \{ rv1[m+j-1] * rv2[j], j=1 \text{ to } n1-m+1 \}, \text{ for } m = 1 \text{ to } no$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

All vectors must be compact.

## **cdotpr**

## *Complex Dot Product*

### **C USAGE:**

```
float cv1[][2], cv2[][2], cso[2];
long inc1, inc2, n;
cdotpr_(cv1, &inc1, cv2, &inc2, cso, &n);
-or-
cdotpr(cv1, inc1, cv2, inc2, cso, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cv2(), cso
integer*4 inc1, inc2, n
call cdotpr(cv1, inc1, cv2, inc2, cso, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector 1
inc1	(inp) - Vector increment for cv1
cv2	(inp) - Complex input vector 2
inc2	(inp) - Vector increment for cv2
cso	(out) - Complex scalar output
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Two complex vectors are multiplied together and summed into the scalar cso.

### **MATHEMATICAL EQUIVALENT:**

$$cso = \text{SUM}\{ cv1[i*inc1] * cv2[i*inc2] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cfft**

*Complex FFT, in Place - Fwd and Inv - Sizes to 1024 Cmplx*

### **C USAGE:**

```
float cvio[][2];
long n, flg;
cfft_(cvio, &n, &flg);
-or-
cfft(cvio, n, flg);
```

### **FORTRAN USAGE:**

```
complex cvio()
integer*4 n, flg
call cfft(cvio, n, flg)
```

### **ARGUMENT DESCRIPTIONS:**

cvio	(i/o) - Complex input vector
n	(inp) - FFT size (number of complex points)
flg	(inp) - Flag selecting transform direction
	+1 => forward, -1 => inverse

### **SUBROUTINE DESCRIPTION:**

The forward or inverse complex FFT of the input vector is computed in place.

### **MATHEMATICAL EQUIVALENT:**

for  $k=0$  to  $m-1$ :

$$Y(m) = \text{SUM} \{ x[k] * [W^{**}(m*k)] \}, \text{ for } m=0 \text{ to } n-1$$

where  $x = \text{cvio}$ ,  $Y = \text{cvio}$ ,  $j = \text{SQRT} \{-1\}$   
 $W = \text{EXP} \{-j * 2\pi/n\}$  for  $\text{flg} = +1$   
 $W = \text{EXP} \{+j * 2\pi/n\}$  for  $\text{flg} = -1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Size  $n$  must be a power of 2 in the range  $8 \leq n \leq 1024$ .
- 2) Input  $\text{cvio}$  must be a compact vector (stride = 2), aligned on an 8-byte boundary.
- 3)  $\text{fftwts}$  must be called before calling  $\text{cfft}$ , with size  $>$  FFT size.

## **cfft**

*Complex Forward FFT - Sizes to 1048576 (1 M) Cmplx*

### **C USAGE:**

```
float cv1[][2], cvo[][2];
long incl, inco, n;
cfft_(cv1, &incl, cvo, &inco, &n);
-or-
cfft(cv1, incl, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cvo()
integer*4 incl, inco, n
call cfft(cv1, incl, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Input vector increment
cvo	(out) - Complex output vector
inco	(inp) - Output vector increment
n	(inp) - FFT size (number of complex points)

### **SUBROUTINE DESCRIPTION:**

The forward complex FFT of the input vector is computed and stored in the output vector.

### **MATHEMATICAL EQUIVALENT:**

for  $k=0$  to  $m-1$ :  
$$Y(m) = \text{SUM} \{ x[k] * W^{m*k} \}, \text{ for } m=0 \text{ to } n-1$$
  
where  $x = cv1$ ,  $Y = cvo$ ,  $j = \text{SQRT} \{-1\}$ ,  $W = \text{EXP} \{-j*2\pi/n\}$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Size  $n$  must be a power of 2 in the range  $8 \leq n \leq 1048576$ .
- 2) Input and output vectors  $cv1$  and  $cvo$  must be aligned on 8-byte boundaries.
- 3) `fftwts` must be called before calling `cfft`, with size  $\geq$  FFT size.
- 4) For sizes  $> 1024$ , input vector  $cv1$  is overwritten.

## **cfft**

*Complex Inverse FFT - Sizes to 1048576 (1 M) Cmplx*

### **C USAGE:**

```
float cv1[][2], cvo[][2];
long incl, inco, n;
cfft_(cv1, &incl, cvo, &inco, &n);
-or-
cfft(cv1, incl, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cvo()
integer*4 incl, inco, n
call cfft(cv1, incl, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Input vector increment
cvo	(out) - Complex output vector
inco	(inp) - Output vector increment
n	(inp) - IFFT size (number of complex points)

### **SUBROUTINE DESCRIPTION:**

The inverse complex FFT of the input vector is computed and stored in the output vector.

### **MATHEMATICAL EQUIVALENT:**

for  $k=0$  to  $m-1$ :  
 $Y(m) = \text{SUM} \{ x[k] * [W^{**}(m*k)] \}$ , for  $m=0$  to  $n-1$   
where  $x = cv1$ ,  $Y = cvo$ ,  $j = \text{SQRT} \{-1\}$ ,  $W = \text{EXP}\{+j*2\pi/n\}$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Size  $n$  must be a power of 2 in the range  $8 \leq n \leq 1048576$
- 2) Input and output vectors  $cv1$  &  $cvo$  must be aligned on 8-byte boundaries.
- 3) `fftwts` must be called before calling `cfft`, with size  $\geq$  FFT size.
- 4) For sizes  $> 1024$ , input vector  $cv1$  is overwritten.

## **cfftsc**

*Complex FFT Scale*

### **C USAGE:**

```
float cvio[][2];
long n;
cfftsc_(cvio, &n);
-or-
cfftsc(cvio, n);
```

### **FORTRAN USAGE:**

```
complex cvio()
integer*4 n
call cfftsc(cvio, n)
```

### **ARGUMENT DESCRIPTIONS:**

cvio	(i/o) - Complex input/output vector
n	(inp) - Number of complex points in cvio

### **SUBROUTINE DESCRIPTION:**

Each point in the input vector is multiplied by the real scalar  $1/n$  and stored back into the vector. This properly scales the output of a complex FFT.

### **MATHEMATICAL EQUIVALENT:**

$cvio[i] = cvio[i] * 1/n$ , for  $i=0$  to  $n-1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

Vector cvio must be compact.

## **cft2fc**

*Two Dimensional Complex Forward      FFT - Column Compact*

### **C USAGE:**

```
float cv1[][2], cvo[][2];
long nr, nc;
cft2fc_(cv1, cvo, &nr, &nc);
-or-
cft2fc(cv1, cvo, nr, nc);
```

### **FORTRAN USAGE:**

```
complex cv1(), cvo()
integer*4 nr, nc
call cft2fc(cv1, cvo, nr, nc)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
cvo	(out) - Complex output vector
nr	(inp) - Number of rows in cv1
nc	(inp) - Number of columns in cv1

### **SUBROUTINE DESCRIPTION:**

The two dimensional forward complex FFT of the input vector is computed and stored in the output vector.  
The columns are assumed to be compact.

### **MATHEMATICAL EQUIVALENT:**

```
for i = 0 to nr-1:
    cfft(cvl, 2*nr, cvo, 2*nr, nc)
for i = 0 to nc-1:
    cffif(cvo, 2, cvo, 2, nr);
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Input data is assumed to be in column order (FORTRAN)
- 2) Input and output vectors cv1 & cvo must be aligned on 8-byte boundaries.
- 3) fftwts, must be called before calling cft2fc, with size  $\geq \text{MAX}(\text{nr}, \text{nc})$ .
- 4) Size is restricted to 1024 x 1024.



## **cft2fr**

### *Two Dimensional Complex Forward FFT - Row Compact*

#### **C USAGE:**

```
float cv1[][2], cvo[][2];
long nr, nc;
cft2fr_(cv1, cvo, &nr, &nc);
-or-
cft2fr(cv1, cvo, nr, nc);
```

#### **FORTRAN USAGE:**

```
complex cv1(), cvo()
integer*4 nr, nc
call cft2fr(cv1, cvo, nr, nc)
```

#### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
cvo	(out) - Complex output vector
nr	(inp) - Number of rows in cv1
nc	(inp) - Number of columns in cv1

#### **SUBROUTINE DESCRIPTION:**

The two dimensional forward complex FFT of the input vector is computed and stored in the output vector. The columns are assumed to be compact.

#### **MATHEMATICAL EQUIVALENT:**

```
for i = 0 to nr-1:
    cfft(cvl, 2, cvo, 2, nc)
for i = 0 to nc-1:
    cfft(cvo, 2*nc, cvo, 2*nc, nr);
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Input data is assumed to be in column order (FORTRAN)
- 2) Input and output vectors cv1 & cvo must be aligned on 8-byte boundaries.
- 3) fftwts, must be called before calling cft2fc, with size  $\geq \text{MAX}(\text{nr}, \text{nc})$ .
- 4) Size is restricted to 1024 x 1024.

## **cft2ic**

### *Two Dimensional Complex Inverse FFT - Column Compact*

#### **C USAGE:**

```
float cv1[][2], cvo[][2];
long nr, nc;
cft2ic_(cv1, cvo, &nr, &nc);
-or
cft2ic(cv1, cvo, nr, nc);
```

#### **FORTRAN USAGE:**

```
complex cv1(), cvo()
integer*4 nr, nc
call cft2ic(cv.1, cvo, nr, nc)
```

#### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
cvo	(out) - Complex output vector
nr	(inp) - Number of rows in cv1
nc	(inp) - Number of columns in cv1

#### **SUBROUTINE DESCRIPTION:**

The two dimensional forward complex FFT of the input vector is computed and stored in the output vector. The columns are assumed to be compact.

#### **MATHEMATICAL EQUIVALENT:**

```
for i = 0 to nr-1:
    cffti(cv1, 2*nr, cvo, 2*nr, nc);
for i = 0 to nc-1:
    cffti(cvo, 2, cvo, 2, nr);
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Input data is assumed to be in column order (FORTRAN).
- 2) Input and output vectors cv1 & cvo must be aligned on 8-byte boundaries.
- 3) fftwts must be called before calling cft2ic, with size  $\geq \text{MAX}(\text{nr}, \text{nc})$ .
- 4) Size is restricted to 1024 x 1024.

## **cft2ir**

### *Two Dimensional Complex Inverse FFT - Row Compact*

#### **C USAGE:**

```
float cv1[][2], cvo[][2];
long nr, nc;
cft2ir_(cv1, cvo, &nr, &nc);
-or
cft2ir(cv1, cvo, nr, nc);
```

#### **FORTRAN USAGE:**

```
complex cv1(), cvo()
integer*4 nr, nc
call cft2ir(cv.1, cvo, nr, nc)
```

#### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
cvo	(out) - Complex output vector
nr	(inp) - Number of rows in cv1
nc	(inp) - Number of columns in cv1

#### **SUBROUTINE DESCRIPTION:**

The two dimensional forward complex FFT of the input vector is computed and stored in the output vector.  
The columns are assumed to be compact.

#### **MATHEMATICAL EQUIVALENT:**

```
for i = 0 to nr-1:
    cffti(cv1, 2, cvo, 2, nc);
for i = 0 to nc-1:
    cffti(cvo, 2*nc, cvo, 2*nc, nr);
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Input data is assumed to be in column order (FORTRAN).
- 2) Input and output vectors cv1 & cvo must be aligned on 8-byte boundaries.
- 3) fftwts must be called before calling cft2ic, with size  $\geq \text{MAX}(\text{nr}, \text{nc})$ .
- 4) Size is restricted to 1024 x 1024.

## convd

## Convolution with Decimation

### C USAGE:

```
float rv1[], rv2[], rvo[];
long inc1, inc2, inco, no, n2, ndf;
convd_(rv1, &inc1, rv2, &inc2, rvo, &inco, &no, &n2, &ndf);
-or-
convd(rv1, inc1, rv2, inc2, rvo, inco, no, n2, ndf);
```

### FORTRAN USAGE:

```
real rv1(), rv2(), rvo()
integer*4 inc1, inc2, no, n2, ndf
call convd(rv1, inc1, rv2, inc2, rvo, inco, no, n2, ndf)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
no	(inp) - Number of points in output vector
n2	(inp) - Number of points in vector rv2 (kernel)
ndf	(inp) - Decimation factor

### SUBROUTINE DESCRIPTION:

Performs a convolution of rv1 using rv2 as the kernel, placing the resulting result in rvo, decimated by the factor specified by ndf. A corrolation can be performed by swapping the elements of rv2 end for end. ie (move 1st element to last, last element to first).

### MATHEMATICAL EQUIVALENT:

for k=0 to no-1:  
$$rvo[k*inco] = \text{SUM} \{ rv1 [inc; 1 *(m + k*ndf)] * rv2[m*inc2] \}; \text{ for } m = 0 \text{ to } n2-1$$

### RESTRICTIONS & SPECIAL CONDITIONS:

rv1 must contain at least (no-1)\*ndf + n2 elements.

## **convrr**

*Convolution Real to Real*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, no, n2;

convrr_(rv1, &incl, rv2, &inc2, rvo, &inco, &no, &n2)
-or-
convrr(rv1, incl, rv2, inc2, rvo, inco, no, n2)
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, no, n2
call convrr(rv1, incl, rv2, inc2, rvo, inco, no, n2)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
no	(inp) - Number of points in output vector
n2	(inp) - Number of points in vector rv2 (kernel)

### **SUBROUTINE DESCRIPTION:**

Performs a convolution of rv1 using rv2 as the kernel, placing the resulting result in rvo. A corrolation can be performed by swapping the elements of rv2 end for end. ie (move 1st element to last, last element to first). The convolution is performed using all values in the kernel, so this routine will work with non-symetrical kernels.

### **MATHEMATICAL EQUIVALENT:**

for k=0 to no-1:  
$$rvo[k*inco] = \text{SUM} \{ rv1 [inc; 1 *(m + k)] * rv2[m*inc2] \}; \text{ for } m = 0 \text{ to } n2-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

rv1 must contain at least (no-1) + n2 elements.

## **convrrs**

### *Convolution Real to Real Symmetrical*

#### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, no, n2;

convrrs_(rv1, &incl, rv2, &inc2, rvo, &inco, &no, &n2)
-or-
convrrs(rv1, incl, rv2, inc2, rvo, inco, no, n2)
```

#### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, no, n2
call convrrs(rv1, incl, rv2, inc2, rvo, inco, no, n2)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
no	(inp) - Number of points in output vector
n2	(inp) - Number of points in vector rv2 (kernel)

#### **SUBROUTINE DESCRIPTION:**

Performs a convolution of rv1 using rv2 as the symmetrical kernel, placing the resulting result in rvo. A correlation can be performed by swapping the elements of rv2 end for end. ie (move 1st element to last, last element to first). This convolution is performed in significantly less time than convrr,

#### **MATHEMATICAL EQUIVALENT:**

for k=0 to no-1:  
$$rvo[k*inco] = \text{SUM} \{ rv1 [inc; 1 *(m + k)] * rv2[m*inc2] \}; \text{ for } m = 0 \text{ to } n2-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

rv1 must contain at least (no-1) + n2 elements.  
rv2 must be symmetrical (even or odd)

## **cpow**

## *Complex Vector Power with Add*

### **C USAGE:**

```
float cv1[][2], rs1, rv2[], rs2, rvo[];
long inc1, inc2, inco, n;
cpow_(cv1, &inc1, &rs1, rv2, &inc2, &rs2, rvo, &inco, n);
-or-
cpow(cv1, inc1, &rs1, rv2, inc2, &rs2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1()
real*4 rs1, rv2(), rs2, rvo()
integer*4 inc1, inc2, inco, n
call cpow(cv1, inc1, rs1, rv2, inc2, rs2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
inc1	(inp) - Vector increment for cv1
rs1	(inp) - Real scalar 1
rv2	(inp) - Real input vector
inc2	(inp) - Vector increment for rv2
rs2	(inp) - Real scalar 2
rvo	(out) - Real output vector
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The magnitude squared of complex vector cv1 is scaled by real rs1 and summed with real vector rv1 scaled by rs2.

### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    rvo[i*inco] = cv1[i*inc1] * CONJ{ cv1[i*inc 1]} * rs1 + rv2[i*inc2] * rs2
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **crvdiv**

## *Complex Vector Real Vector Divide*

### **C USAGE:**

```
float cv1[][2], rv2[], cvo[][2];
long inc1, inc2, inco, n;
crvdiv_(cv1, &inc1, rv2, &inc2, cvo, &inco, &n);
-or-
crvdiv(cv1, inc1, rv2, inc2, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cvo()
real*4 rv2()
integer*4 inc1, inc2, inco, n
call crvdiv(cv1, inc1, rv2, inc2, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
inc1	(inp) - Input vector increment
rv2	(inp) - Real input vector
inc2	(inp) - Real vector increment
cvo	(out) - Complex output vector
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Each element of the complex input vector, cv1, is divided by the corresponding element of the real vector, rv2, and copied to the complex output vector, cvo.

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco]=cv1[i*inc1] / rv2[i*inc2], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **crvmul**

## *Complex Vector Real Vector Multiply*

### **C USAGE:**

```
float cv1[][2], rv2[], cvo[][2];
long incl, inc2, inco, n;
crvmul_(cv1, &incl, rv2, &inc2, cvo, &inco, &n);
-or-
crvmul(cv1, incl, rv2, inc2, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cvo()
real*4 rv2()
integer*4 incl, inc2, inco, n
call crvmul(cv1, incl, rv2, inc2, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Complex vector increment
rv2	(inp) - Real input vector
inc2	(inp) - Real vector increment
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Each element of the complex input vector cv1 is multiplied by the corresponding element of the real vector, rv2, and copied to the complex output vector cvo.

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco] = cv1[i*incl] * rv2[i*inc2], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cspec**

## *Accumulating Cross Spectrum*

### **C USAGE:**

```
float cv1[][2], cv2[][2], cvo[][2];
long n;
cspec_(cv1, cv2, cvo, &n);
-or-
cspec(cv1, cv2, cvo, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cv2(), cvo()
integer*4 n
call cspec(cv1, cv2, cvo, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector 1
cv2	(inp) - Complex input vector 2
cvo	(i/o) - Complex input/output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The complex conjugate of the first input vector, cv1, is multiplied by the second input vector, cv2, and summed with the i/o vector cvo. (This routine is included only for backwards compatibility with other subroutine libraries. For new programs, use cvcm1a.)

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i] = \text{CONJ}\{ cv1[i] \} * cv2[i] + cvo[i], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

All vectors must be compact.

## **cvabs**

## *Complex Vector Absolute Value*

### **C USAGE:**

```
float cv1[][2], rvo[];
long incl, inco, n;
cvabs_(cv1, &incl, rvo, &inco, &n);
-or-
cvabs(cv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1()
real*4 rvo()
integer*4 incl, inco, n
call cvabs(cv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Complex vector increment
rvo	(out) - Real output vector
inco	(inp) - Real vector increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The absolute value of each element of the complex input vector is copied to the real output vector, rvo. The absolute value is computed from the square root of the magnitude squared of each element.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{SQRT}\{ cv1[i*incl] * cv1[i*incl] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvadd**

## *Complex Vector Add*

### **C USAGE:**

```
float cv1[][2], cv2[][2], cvo[][2];
long inc1, inc2, inco, n;
cvadd_(cv1, &inc1, cv2, &inc2, cvo, &inco, &n);
-or-
cvadd(cv1, inc1, cv2, inc2, cvo, inco, n);
```

### **FORTTRAN USAGE:**

```
complex cv1(), cv2(), cvo()
integer*4 inc1, inc2, inco, n
call cvadd(cv1, inc1, cv2, inc2, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector 1
inc1	(inp) - Vector increment for cv1
cv2	(inp) - Complex input vector 2
inc2	(inp) - Vector increment for cv2
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The elements of the complex input vectors are added together and stored to the complex output vector cvo.

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco] = cv1[i*inc1] + cv2[i*inc2], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvcml**

## *Complex Vector Conjugate Multiply*

### **C USAGE:**

```
float cv1[][2], cv2[][2], cvo[][2];
long inc1, inc2, inco, n;
cvcml_(cv1, &inc1, cv2, &inc2, cvo, &inco, &n);
-or-
cvcml(cv1, inc1, cv2, inc2, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cv2(), cvo()
integer*4 inc1, inc2, inco, n
call cvcml(cv1, inc1, cv2, inc2, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector 1
inc1	(inp) - Vector increment for cv1
cv2	(inp) - Complex input vector 2
inc2	(inp) - Vector increment for cv2
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Elements of the second complex vector are multiplied by the complex conjugate of the first complex vector. The results are written to the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco] = \text{CONJ}\{ cv1[i*inc1] \} * cv2[i*inc2], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvcmla**

### *Complex Vector Conjugate Multiply with Add*

#### **C USAGE:**

```
float cv1[][2], cv2[][2], cv3[][2], cvo[][2];
long inc1, inc2, inc3, inco, n;
cvcmla_(cv1, &inc1, cv2, &inc2, cv3, &inc3, cvo, &inco, &n);
-or-
cvcmla(cv1, inc1, cv2, inc2, cv3, inc3, cvo, inco, n);
```

#### **FORTRAN USAGE:**

```
complex cv1(), cv2(), cv3(), cvo()
integer*4 inc1, inc2, inc3, inco, n
call cvcmla(cv1, inc1, cv2, inc2, cv3, inc3, cvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector 1
inc1	(inp) - Vector increment for cv1
cv2	(inp) - Complex input vector 2
inc2	(inp) - Vector increment for cv2
cv3	(inp) - Complex input vector 3
inc3	(inp) - Vector increment for cv3
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

The complex conjugate of vector cv 1 is multiplied by the complex vector cv2, that result is added to the elements from complex vector cv3 and stored in complex vector cvo.

#### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco] = \text{CONJ}\{ cv1[i*inc1] \} * cv2[i*inc2] + cv3[i*inc3], \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvcomb**

### *Form Complex Vector from Two Real Vectors*

#### **C USAGE:**

```
float rv1[], rv2[];
float cvo[][2];
long inc1, inc2, inco, n;
cvcomb_(rv1, &inc1, rv2, &inc2, cvo, &inco, &n);
-or-
cvcomb(rv1, inc1, rv2, inc2, cvo, inco, n);
```

#### **FORTRAN USAGE:**

```
real rv1(), rv2()
complex cvo()
integer*4 inc2, inc2, inco, n
call cvcomb(rv1, inc1, rv2, inc2, cvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector (r)
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector (i)
inc2	(inp) - Vector increment for rv2
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

Sets the desired elements of the complex array to a complex value with the real part equal to elements of the first vector and the imaginary parts equal to elements of the second vector.

#### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    cvo[i*inco] = rv1[i*inc1], rv2[i*inc2]
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvconj**

## *Complex Vector Conjugate*

### **C USAGE:**

```
float cv1[][2], cvo[][2];
long incl, inco, n;
cvconj_(cv1, &incl, cvo, &inco, &n);
-or-
cvconj(cv1, incl, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cvo()
integer*4 incl, inco, n
call cvconj(cv1, incl, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Vector increment for cv1
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Each element of the complex input vector is copied to the complex output vector, cvo, with the imaginary part negated (complex conjugate).

### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    cvo(i*inco).r = cv1(i*incl).r
    cvo(i*inco).i = -cv1 (i*inc l).i
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **cvcsml**

## *Complex Vector Complex Scalar Multiply*

### **C USAGE:**

```
float cv1[][2], cs1[2], cvo[][2];
long incl, inco, n;
cvcsml_(cv1, &incl, cs1, cvo, &inco, n);
-or-
cvcsml(cv1, incl, cs1, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cs1, cvo()
integer*4 incl, inco, n
call cvcsml(cv1, incl, cs1, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Vector increment for cv1
cs1	(inp) - Complex scalar
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Each element of the complex input vector is multiplied by the complex scalar cs 1 and is then copied to the complex output vector cvo.

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco] = cs1 * cv1[i*incl], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvdiv**

### *Complex Vector Complex Vector Division*

#### **C USAGE:**

```
float cv1[][2], cv2[][2], cvo[][2];
long inc1, inc2, inco, n;
cvdiv_(cv1, &inc1, cv2, &inc2, cvo, &inco, &n);
-or-
cvdiv(cv1, inc1, cv2, inc2, cvo, inco, n);
```

#### **FORTRAN USAGE:**

```
complex cv1(), cv2(), cvo()
integer*4 inc1, inc2, inco, n
call cvdiv(cv1, inc1, cv2, inc2, cvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector 1
inc1	(inp) - Complex vector increment for cv1
cv2	(inp) - Complex input vector 2
inc2	(inp) - Complex vector increment for cv2
cvo	(out) - Complex output vector
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

Elements of the first complex input vector, cv1, are divided by elements of the second complex input vector, cv2, and copied to the complex output vector, cvo.

#### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco]=cv1[i*inc1] / cv2[i*inc2], \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvexp**

## *Complex Vector Exponential*

### **C USAGE:**

```
float rv1[], cvo[][2];
long incl, inco, n;
cvexp_(rv1, &incl, cvo, &inco, &n);
-or-
cvexp(rv1, incl, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1()
complex cvo()
integer*4 incl, inco, n
call cvexp(rv1, incl, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
cvo	(out) - Complex output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes complex exponential of each element of real input vector, with input values expressed in radians.

### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
  cvo[i*inco].r = COS{ rv1[i*incl] }
  cvo[i*inco].i = SIN{ rv1[i*incl] }
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvfill**

*Set a Complex Vector to a Complex Scalar*

### **C USAGE:**

```
float cvo[][2], cs1[2];
long inco, n;
cvfill_(cs1, cvo, &inco, &n);
-or-
cvfill(cs1, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cvo(), cs1
integer*4 inco, n
call cvfill(cs1, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cs1	(inp) - Complex scalar
cvo	(out) - Complex output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of points in output vector

### **SUBROUTINE DESCRIPTION:**

Fills vector cvo with the scalar cs1.

### **MATHEMATICAL EQUIVALENT:**

$cv[i*inco]=cs1$ , for  $i=0$  to  $n-1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvma**

### *Complex or Conjugate Vector Multiply with Add*

#### **C USAGE:**

```
float cv1[][2], cv2[][2], cv3[][2], cvo[][2];
long inc1, inc2, inc3, inco, n, flg;
cvma_(cv1, &inc1, cv2, &inc2, cv3, &inc3, cvo, &inco, &n, &flg);
-or-
cvma(cv1, inc1, cv2, inc2, cv3, inc3, cvo, inco, n, flg);
```

#### **FORTRAN USAGE:**

```
complex cv1(), cv2(), cv3(), cvo()
integer*4 inc1, inc2, inc3, inco, n, flg
call cvma(cv1, inc1, cv2, inc2, cv3, inc3, cvo, inco, n, flg)
```

#### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector 1
inc1	(inp) - Vector increment for cv1
cv2	(inp) - Complex input vector 2
inc2	(inp) - Vector increment for cv2
cv3	(inp) - Complex input vector 3
inc3	(inp) - Vector increment for cv3
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
n	(inp) - Number of samples to process
flg	(inp) - Flag to specify complex or conjugate multiply
	flg >= 0 - Complex Multiply
	flg < 0 - Conjugate Multiply

#### **SUBROUTINE DESCRIPTION:**

Elements of the first two input complex vectors are multiplied together as specified by flg and added to the third complex input vector. Complex results are written to the complex output vector.

#### **MATHEMATICAL EQUIVALENT:**

if flg >= 0 then  $cvo[i*inco] = cv1[i*inc1] * cv2[i*inc2] + cv3[i*inc3]$ , for  $i=0$  to  $n-1$   
else  $cvo(i*inco) = \text{CONJ}\{ cv1[i*inc1] \} * cv2[i*inc2] + cv3[i*inc3]$ , for  $i=0$  to  $n-1$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvmags**

## *Complex Vector Magnitude Squared*

### **C USAGE:**

```
float cv1[][2], rvo[];
long incl, inco, n;
cvmags_(cv1, &incl, rvo, &inco, &n);
-or-
cvmags(cv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1()
real*4 rvo()
integer*4 incl, inco, n
call cvmags(cv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Vector increment for cv 1
rvo	(out) - Real output vector
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The squares of the real and imaginary parts of each element in the complex input vector are summed and returned in the corresponding element of the real output vector.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i * inco] = cv1[i * incl].r^2 + cv1[i * incl].i^2, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvmexp**

## *Complex Vector Exponential with Multiply*

### **C USAGE:**

```
float rv1[], rv2[], cvo[][2];
long inc1, inc2, inco, n;
cvmexp_(rv1, &inc1, rv2, &inc2, cvo, &inco, &n);
-or-
cvmexp(rv1, inc1, rv2, inc2, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
real*4 rv1(), rv2()
complex cvo()
integer*4 inc1, inc2, inco, n
call cvmexp(rv1, inc1, rv2, inc2, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Real vector increment 1
rv2	(inp) - Second real input vector
inc2	(inp) - Second input vector increment
cvo	(out) - Complex vector output
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The cosine of each element of the real input vector, rv1, is scaled by the corresponding element of rv2 and assigned to the real part of the complex output vector, cvo. The sine of each element of the real input vector, rv1, is scaled by rv2 and assigned to the imaginary part of the complex output vector, cvo.

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco] = (rv2[i*inc2] * \cos\{rv1[i*inc1]\}, rv2[i*inc2] * \sin\{rv1[i*inc1]\}), \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvmgsa**

### *Complex Vector Magnitude Squared with Add*

#### **C USAGE:**

```
float cv1[][2], rv2[], rvo[];
long incl, inc2, inco, n;
cvmgsa_(cv1, &incl, rv2, &inc2, rvo, &inco, &n);
-or-
cvmgsa(cv1, incl, rv2, inc2, rvo, inco, n);
```

#### **FORTRAN USAGE:**

```
complex cv1()
real*4 rv2(), rvo()
integer*4 incl, inc2, inco, n
call cvmgsa(cv1, incl, rv2, inc2, rvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Complex vector increment
rv2	(inp) - Real input vector
inc2	(inp) - Real vector increment
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

Each element of the complex input vector cv 1 is multiplied by its own conjugate and the corresponding element of the real vector rv2 is added to it. The result is placed in the real vector rvo.

#### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = cv1[i*incl] * \text{CONJ}\{ cv1[i*incl] + rv2[i*inc2] \} , \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## cvml

## Complex Vector Multiply

### C USAGE:

```
float cv1[][2], cv2[][2], cvo[][2];
long inc1, inc2, inc3, inco, n;
cvmla_(cv1, &inc1, cv2, &inc2, cvo, &inco, &n);
-or-
cvmla(cv1, inc1, cv2, inc2, cvo, inco, n);
```

### FORTRAN USAGE:

```
complex cv1(), cv2(), cvo()
integer*4 inc1, inc2, inco, n
call cvmla(cv1, inc1, cv2, inc2, cvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

cv1	(inp) - Complex input vector 1
inc1	(inp) - Vector increment for cv1
cv2	(inp) - Complex input vector 2
inc2	(inp) - Vector increment for cv2
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
n	(inp) - Number of samples to process

### SUBROUTINE DESCRIPTION:

Elements of the first two input complex vectors are multiplied together. Complex results are written to the complex output vector.

### MATHEMATICAL EQUIVALENT:

$$cvo[i*inco] = cv1[i*inc1] * cv2[i*inc2], \text{ for } i=0 \text{ to } n-1$$

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## cvmla

## Complex Vector Multiply with Add

### C USAGE:

```
float cv1[][2], cv2[][2], cv3[][2], cvo[][2];
long inc1, inc2, inc3, inco, n;
cvmla_(cv1, &inc1, cv2, &inc2, cv3, &inc3, cvo, &inco, &n);
-or-
cvmla(cv1, inc1, cv2, inc2, cv3, inc3, cvo, inco, n);
```

### FORTTRAN USAGE:

```
complex cv1(), cv2(), cv3(), cvo()
integer*4 inc1, inc2, inc3, inco, n
call cvmla(cv1, inc1, cv2, inc2, cv3, inc3, cvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

cv1	(inp) - Complex input vector 1
inc1	(inp) - Vector increment for cv1
cv2	(inp) - Complex input vector 2
inc2	(inp) - Vector increment for cv2
cv3	(inp) - Complex input vector 3
inc3	(inp) - Vector increment for cv3
cv0	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
n	(inp) - Number of samples to process

### SUBROUTINE DESCRIPTION:

Elements of the first two input complex vectors are multiplied together and added to the third complex input vector. Complex results are written to the complex output vector.

### MATHEMATICAL EQUIVALENT:

$$cvo[i*inco] = cv1[i*inc1] * cv2[i*inc2] + cv3[i*inc3], \text{ for } i=0 \text{ to } n-1$$

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## **cvmov**

## *Complex Vector Move*

### **C USAGE:**

```
float cv1[][2], cvo[][2];
long incl, inco, n;
cvmov_(cv1, &incl, cvo, &inco, &n);
-or-
cvmov(cv1, incl, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cvo()
integer*4 incl, inco, n
call cvmov(cv1, incl, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Vector increment for cv1
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
n	(inp) - Number of points to move

### **SUBROUTINE DESCRIPTION:**

Elements from the input vector cv1 are copied to the output vector cvo.

### **MATHEMATICAL EQUIVALENT:**

$cvo[i*inco]=cv1[i*incl]$ , for  $i=0$  to  $n-1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvmul**

### *Complex Vector Multiply or Conjugate Multiply*

#### **C USAGE:**

```
float cv1[][2], cv2[][2], cvo[][2];
long incl, inc2, inco, n, flg;
cvmul_(cv1, &incl, cv2, &inc2, cvo, &inco, &n, &flg);
-or-
cvmul(cv1, incl, cv2, inc2, cvo, inco, n, flg);
```

#### **FORTRAN USAGE:**

```
complex cv1(), cv2(), cvo()
integer*4 incl, inc2, inco, n, flg
call cvmul(cv1, incl, cv2, inc2, cvo, inco, n, flg)
```

#### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector 1
incl	(inp) - Vector increment 1
cv2	(inp) - Complex input vector 2
inc2	(inp) - Vector increment 2
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
n	(inp) - Number of samples to process
flg	(inp) - Flag: < 0 = conjugate, otherwise normal

#### **SUBROUTINE DESCRIPTION:**

The two complex vectors are multiplied and the result written to the output vector. The flg parameter controls whether or not the complex conjugate of the first vector is used.

#### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    if flg<0 then cvo[i*inco]=CONJ{ cv1[i*incl] } *cv2[i*inc2]
    else cvo(i*inco) = cv1[i*incl] * cv2[i*inc2]
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvneg**

## *Complex Vector Negate*

### **C USAGE:**

```
float cv1[][2], cvo[][2];
long incl, inco, n;
cvneg_(cv1, &incl, cvo, &inco, &n);
-or-
cvneg(cv1, incl, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cvo()
integer*4 incl, inco, n
call cvneg(cv1, incl, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Vector increment for cv1
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Each element of the complex input vector cv1 is negated and copied to the complex Output vector cvo.

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco] = -cv1[i*incl], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvphas**

## *Complex Vector Phase*

### **C USAGE:**

```
float cv1[][2], rvo[];
long incl, inco, n;
cvphas_(cv1, &incl, rvo, &inco, &n);
-or-
cvphas(cv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), rvo()
integer*4 incl, inco, n
call cvphas(cv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Input vector increment
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The phase or argument (theta from polar form) of each element in the complex input vector is copied to the real output vector rvo.

### **MATHEMATICAL EQUIVALENT:**

$$\text{rvo}[i*\text{inco}] = \text{ARCTAN}\{ \text{cv1}[i*\text{incl}].\text{r} / \text{cv1}[i*\text{incl}].\text{i} \}$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvrcip**

## *Complex Vector Reciprocal*

### **C USAGE:**

```
float cv1[][2], cvo[][2];
long incl, inco, n;
cvrcip_(cv1, &incl, cvo, &inco, &n);
-or-
cvrcip(cv1, incl, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cvo()
integer*4 incl, inco, n
call cvrcip(cv1, incl, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Input vector increment
cvo	(out) - Complex output vector
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The reciprocal of the elements of the complex input vector cv1 is copied to the complex output vector cvo.

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco] = 1 / cv1[i*incl], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvreal**

*Make a Complex Vector from a Real Vector*

### **C USAGE:**

```
float rv1[], cvo[][2];
long incl, inco, n;
cvreal_(rv1, &incl, cvo, &inco, &n);
-or-
cvreal(rv1, incl, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex rv1(), cvo()
integer*4 incl, inco, n
call cvreal(rv1, incl, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment for rv1
cvo	(out) - Complex output vector
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

A complex vector, cvo, is created from a real vector rv1. The imaginary part of cvo is set to zero.

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco] = (rv1[i*incl], 0), \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **cvsma**

### *Scale Complex Vector and Add to Second Complex Vector*

#### **C USAGE:**

```
float cv1[][2], cs1[2], cv2[][2], cvo[][2];
long incl, inc2, inco, n;
cvsma_(cv1, &incl, cs1, cv2, &inc2, cvo, &inco, &n);
-or-
cvsma(cv1, incl, cal, cv2, inc2, cvo, inco, n);
```

#### **FORTRAN USAGE:**

```
complex cv1(), cs1, cv2(), cvo()
integer*4 incl, inc2, inco, n
call cvsma(cv1, incl, cal, cv2, inc2, cvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Complex vector increment for cv1
cs1	(inp) - Complex scalar
cv2	(inp) - Complex input vector
inc2	(inp) - Complex vector increment for cv2
cvo	(out) - Complex output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points to generate

#### **SUBROUTINE DESCRIPTION:**

Each element of the complex input vector, cv1, is multiplied by the complex scalar, cs1, and then added with each element from complex vector, cv2, then finally, copied to the complex output vector, cvo.

#### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco] = cs1 * cv1[i*incl] + cv2[i*inc2], \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvsmul**

## *Complex Vector Scalar Multiply*

### **C USAGE:**

```
float cv1[][2], rs1, cvo[][2];
long incl, inco, n;
cvsmul_(cv1, &incl, &rs1, cvo, &inco, &n);
-or-
cvsmul(cv1, incl, &rs1, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cvo()
real*4 rs1
integer*4 incl, inco, n
call cvsmul(cv1, incl, rs1, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Input vector increment
rs1	(inp) - Real scalar
cvo	(out) - Complex output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points to generate

### **SUBROUTINE DESCRIPTION:**

Each element of the complex input vector, cv 1, is multiplied by the real scalar, rs 1, and then copied to the complex output vector, cvo.

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco] = rs1 * cv1[i*incl], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvsqrt**

## *Complex Vector Square Root*

### **C USAGE:**

```
float cv1[][2], cvo[][2];
long incl, inco, n;
cvsqrt_(cv1, &incl, cvo, &inco, &n);
-or-
cvsqrt(cv1, incl, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cvo()
integer*4 incl, inco, n
call cvsqrt(cv1, incl, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Input vector increment
cvo	(out) - Complex output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points to generate

### **SUBROUTINE DESCRIPTION:**

The square root of each element of the complex input vector cv1 is copied to the complex output vector cvo. The vector is first converted from rectangular to polar form, i.e., magnitude and angle. The complex square root is computed by taking the square root of the magnitude and half the angle, then converting back to rectangular form.

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco]=SQRT\{ cv1[i*incl] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **cvsub**

## *Complex Vector Subtract*

### **C USAGE:**

```
float cv1[][2], cv2[][2], cvo[][2];
long inc1, inc2, inco, n;
cvsub_(cv1, &inc1, cv2, &inc2, cvo, &inco, &n);
-or-
cvsub(cv1, inc1, cv2, inc2, cvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), cv2(), cvo()
integer*4 inc1, inc2, inco, n
call cvsub(cv1, inc1, cv2, inc2, cvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
inc1	(inp) - Input vector increment
cv2	(inp) - Second complex input vector
inc2	(inp) - Second input vector increment
cvo	(out) - Complex output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points to generate

### **SUBROUTINE DESCRIPTION:**

The elements of the complex input vector cv2 are subtracted from the elements in cv1 and stored to cvo.

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i*inco] = cv1[i*inc1] - cv2[i*inc2], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## deq22

*Difference Equation, 2 Poles, 2 Zeroes*

### C USAGE:

```
float rv1[], rv2[], rvo[];
long incl, inco, n;
deq22_(rv1, &incl, rv2, rvo, &inco, &n);
-or-
deq22(rv1, incl, rv2, rvo, inco, n);
```

### FORTRAN USAGE:

```
real rv1(), rv2(), rvo()
integer*4 incl, inco, n
call deq22(rv1, incl, rv2, rvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - First real input vector
incl	(inp) - First input vector increment
rv2	(inp) - Second real input vector
	Five filter coefficients with increment of one.
rvo	(out) - Real output vector
inco	(out) - Output vector increment
n	(inp) - Number of points to output

### SUBROUTINE DESCRIPTION:

Performs a 2-pole, 2-zero recursive filter difference on vector rv1 using the vector rv2 as the filter. The result is stored in vector rvo.

### MATHEMATICAL EQUIVALENT:

$$\text{rvo}[k] = \text{rv1}[k] * \text{rv2}[0] + \text{rv1}[k-1] * \text{rv2}[1] + \text{rv1}[k-2] * \text{rv2}[2] \\ - \text{rvo}[k-1] * \text{rv2}[3] - \text{rvo}[k-2] * \text{rv2}[4]; \text{ for } k=0 \text{ to } n-1$$

### RESTRICTIONS & SPECIAL CONDITIONS:

Output vector rvo, must not overlay input vector rv1. Previous 2 points in output vector rvo[-2] and rvo[-1] must be initialized prior to calling deq22. deq22 will write n points to rvo starting at rv[0].

## **dmxmla**

### *Double Precision Real Matrix plus Matrix Product*

#### **C USAGE:**

```
double dm1[], dm2[], dmo[];
long incc1, incrl, incc2, incr2, incco, incro, nco, nro, ncr;
dmxmla_(dm1, &incc1, &incrl, dm2, &incc2, &incr2, dmo, &incco, &incro,
&nro, &nco, &ncr);
-or-
dmxmla(dm1, incc1, incrl, dm2, incc2, incr2, dmo, incco, incro, nro, nco,
ncr);
```

#### **FORTRAN USAGE:**

```
real*8 dm1(), dm2(), dmo()
integer*4 incc1, incrl, incc2, incr2, incco, incro, nco, nro, ncr
call dmxmla(dm1, incc1, incrl, dm2, incc2, incr2, dmo, incco, incro, nro,
nco, ncr)
```

#### **ARGUMENT DESCRIPTIONS:**

dm1	(inp) - First real matrix
incc1	(inp) - Sample increment between columns
incrl	(inp) - Sample increment between rows
dm2	(inp) - Second real matrix to move
incc2	(inp) - Sample increment between columns
incr2	(inp) - Sample increment between rows
dmo	(i/o) - Real output matrix
incco	(inp) - Sample increment between columns
incro	(inp) - Sample increment between rows
nro	(inp) - Number of rows in output matrix
nco	(inp) - Number of columns in output matrix
ncr	(inp) - Number of columns in dm1 and rows in dm2

#### **SUBROUTINE DESCRIPTION:**

Increases the value of double precision matrix dmo by the product of the two matrices dm1 and dm2

#### **MATHEMATICAL EQUIVALENT:**

$$dmo = dmo + (dm1 * dm2) \text{ (elements set depend on increments)}$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## dmxmls

### *Double Precision Real Matrix minus Matrix Product*

#### C USAGE:

```
double dm1[], dm2[], dmo[];
long incc1, incrl, incc2, incr2, incco, incro, nco, nro, ncr;
dmxmls_(dm1, &incc1, &incrl, dm2, &incc2, &incr2, dmo, &incco, &incro,
&nro, &nco, &ncr);
-or-
dmxmls(dm1, incc1, incrl, dm2, incc2, incr2, dmo, incco, incro, nro, nco,
ncr);
```

#### FORTRAN USAGE:

```
real*8 dm1(), dm2(), dmo()
integer*4 incc1, incrl, incc2, incr2, incco, incro, nco, nro, ncr
call dmxmls(dm1, incc1, incrl, dm2, incc2, incr2, dmo, incco, incro, nro,
nco, ncr)
```

#### ARGUMENT DESCRIPTIONS:

dm1	(inp) - First real matrix
incc1	(inp) - Sample increment between columns
incrl	(inp) - Sample increment between rows
dm2	(inp) - Second real matrix to move
incc2	(inp) - Sample increment between columns
incr2	(inp) - Sample increment between rows
dmo	(i/o) - Real output matrix
incco	(inp) - Sample increment between columns
incro	(inp) - Sample increment between rows
nro	(inp) - Number of rows in output matrix
nco	(inp) - Number of columns in output matrix
ncr	(inp) - Number of columns in dm1 and rows in dm2

#### SUBROUTINE DESCRIPTION:

Decreases the value of double precision matrix dmo by the product of the two matrices dm1 and dm2

#### MATHEMATICAL EQUIVALENT:

$dmo = dmo - (dm1 * dm2)$  (elements set depend on increments)

#### RESTRICTIONS & SPECIAL CONDITIONS:

None

## **dmxmov**

### *Double Precision Real Matrix Move*

#### **C USAGE:**

```
double dm1[], dmo[];
long incc1, incrl, incco, incro, nco, nro;
dmxmov_(dm1, &incc1, &incrl, dmo, &incco, &incro, &nro, &nco);
-or-
dmxmov(dm1, incc1, incrl, dmo, incco, incro, nro, nco);
```

#### **FORTRAN USAGE:**

```
real*8 dm1(), dmo()
integer*4 incc1, incrl, incco, incro, nco, nro
call dmxmov(dm1, incc1, incrl, dmo, incco, incro, nro, nco)
```

#### **ARGUMENT DESCRIPTIONS:**

dm1	(inp) - First real matrix
incc1	(inp) - Sample increment between columns
incrl	(inp) - Sample increment between rows
dmo	(out) - Real output matrix
incco	(inp) - Sample increment between columns
incro	(inp) - Sample increment between rows
nro	(inp) - Number of rows in output matrix
nco	(inp) - Number of columns in output matrix

#### **SUBROUTINE DESCRIPTION:**

This routine moves double precision real elements from one strided matrix to a second strided matrix.

#### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to nco-1:
    for j=0 to nro-1:
        dmo[i*incro+j*incco]=dm1[i*incrl+j*incc1];
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

It is the intent of this routine to provide a means of manipulating sub-matrices and executing various symmetry operations such as transpose. For simply moving a compact matrix, vmov will provide faster execution.



## dmxmul

### *Double Precision Real Matrix Multiply*

#### C USAGE:

```
double dm1[], dm2[], dmo[];
long incc1, incrl, incc2, incr2, incco, incro, nco, nro, ncr;
dmxmul_(dm1, &incc1, &incrl, dm2, &incc2, &incr2, dmo, &incco, &incro,
&nro, &nco, &ncr);
-or-
dmxmul(dm1, incc1, incrl, dm2, incc2, incr2, dmo, incco, incro, nro, nco,
ncr);
```

#### FORTRAN USAGE:

```
real*8 dm1(), dm2(), dmo()
integer*4 incc1, incrl, incc2, incr2, incco, incro, nco, nro, ncr
call dmxmul(dm1, incc1, incrl, dm2, incc2, incr2, dmo, incco, incro, nro,
nco, ncr)
```

#### ARGUMENT DESCRIPTIONS:

dm1	(inp) - First real matrix
incc1	(inp) - Sample increment between columns
incrl	(inp) - Sample increment between rows
dm2	(inp) - Second real matrix to move
incc2	(inp) - Sample increment between columns
incr2	(inp) - Sample increment between rows
dmo	(out) - Real output matrix
incco	(inp) - Sample increment between columns
incro	(inp) - Sample increment between rows
nro	(inp) - Number of rows in output matrix
nco	(inp) - Number of columns in output matrix
ncr	(inp) - Number of columns in dm1 and rows in dm2

#### SUBROUTINE DESCRIPTION:

Performs a real matrix multiply.

#### MATHEMATICAL EQUIVALENT:

$$dmo = dm1 * dm2 \text{ (elements set depend on increments)}$$

#### RESTRICTIONS & SPECIAL CONDITIONS:

None

## **dotpr**

*Real Dot Product*

### **C USAGE:**

```
float rv1[], rv2[], rso;
long inc1, inc2, n;
dotpr_(rv1, &inc1, rv2, &inc2, &rso, &n);
-or-
dotpr(rv1, inc1, rv2, inc2, &rso, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rso
integer*4 inc1, inc2, n
call dotpr(rv1, inc1, rv2, inc2, rso, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rso	(out) - Real scalar output
n	(inp) - Number of points to output

### **SUBROUTINE DESCRIPTION:**

Two real vectors, rv1 and rv2, are multiplied together and summed into a scalar rso.

### **MATHEMATICAL EQUIVALENT:**

$$rso = \text{SUM} \{ rv1[i*inc1] * rv2[i*inc2] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **envel**

## *Vector Envelope*

### **C USAGE:**

```
float rv1[], rvo[];
long n;
envel_(rv1, rvo, &n);
-or-
envel(rv1, rvo, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 n
call envel(rv1, rvo, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
rvo	(out) - Real output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the envelope of a real vector using the Hilbert Transform.

### **MATHEMATICAL EQUIVALENT:**

$$\begin{aligned} rvo &= \text{HLBRT}\{rv1\} \\ rvo[k] &= \text{SQRT}\{rv1[k]^2 + rvo[k]^2\}, \text{ for } k=0 \text{ to } n-1 \end{aligned}$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) All vectors must be compact.
- 2)  $n=2^M$ ,  $5 \leq M \leq 11$ .

## fftwts

## Create FFT Complex Exponential Tables

### C USAGE:

```
float cvo[][2]
long m, n;
fftwts_(cvo, &m, &n);
-or
fftwts(cvo, m, n);
```

### FORTRAN USAGE:

```
complex cvo()
integer*4 m, n
call fftwts(cvo, m, n)
```

### ARGUMENT DESCRIPTIONS:

cvo	(out) - Complex output vector
m	(inp) - Max FFT size (number of complex points)
n	(inp) - FFT size (number of complex points)

### SUBROUTINE DESCRIPTION:

A table of complex exponentials is computed in output vector cvo, for use by FFT and Hanning/Hamming windowing functions.

### MATHEMATICAL EQUIVALENT:

$$Y(k) = [W^{**}(k)], \text{ for } k=0 \text{ to } n-1$$
$$Y = cvo, j = \text{SQRT}\{-1\}, W = \text{EXP}\{-j*2\pi/n\}$$

### RESTRICTIONS & SPECIAL CONDITIONS:

- 1) The size of m must be a power of 2.
- 2) The size of n must be in the range  $3m/4 \leq n \leq m$ .
- 3) The output vector, cvo, must be a compact vector (stride = 2), aligned on an 8-byte boundary.
- 4) fftwts must be called before calling any Hanning/Hamming or FFr routine, with  $n \geq \text{FFT or weighting size required}$ .

## **fix2n**

*Vector Fix to Two-byte Integer and Round*

### **C USAGE:**

```
float rv1[];
short svo[];
long incl, inco, n;
fix2n_(rv1, &incl, svo, &inco, &n);
-or-
fix2n(rv1, incl, svo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1()
integer*2 svo()
integer*4 incl, inco, n
call fix2n(rv1, incl, svo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
svo	(out) - Two-byte integer output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points to process

### **SUBROUTINE DESCRIPTION:**

Converts the real input vector rv1 to integer by rounding and placing the result in a two-byte integer output vector svo, for n points.

### **MATHEMATICAL EQUIVALENT:**

$$\text{svo}[i*\text{inco}] = \text{rv1}[i*\text{incl}], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

Values in rv1 should be between -32768 and 32767.

## **fix4**

### *Vector Fix to Four-byte Integer and Truncate*

#### **C USAGE:**

```
float rv1[];  
long ivo[];  
long incl, inco, n;  
fix4_(rv1, &incl, ivo, &inco, &n);  
-or-  
fix4(rv1, incl, ivo, inco, n);
```

#### **FORTRAN USAGE:**

```
real rv1() integer*4 ivo integer*4 incl, inco, n call fix4(rv1, incl,  
ivo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Real vector increment
ivo	(out) - Four-byte integer output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points to process

#### **SUBROUTINE DESCRIPTION:**

Converts the real input vector rv1 to integer by truncating, and places the result in four-byte integer output vector ivo, for n points.

#### **MATHEMATICAL EQUIVALENT:**

$$\text{ivo}(i*\text{inco}) = \text{rv1}(i*\text{incl})$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

Values in rv1 should be between  $-2^{32}$  and  $+2^{32}-1$ .

## **fix4n**

*Vector Fix to Four-byte Integer and Round*

### **C USAGE:**

```
float rv1[];  
long ivo[];  
long incl, inco, n;  
fix4n_(rv1, &incl, ivo, &inco, &n);  
-or-  
fix4n(rv1, incl, ivo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1()  
integer*4 ivo()  
integer*4 incl, inco, n  
call fix4n(rv1, incl, ivo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
ivo	(out) - Four-byte integer output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points to output

### **SUBROUTINE DESCRIPTION:**

Converts the real input vector rv1 to integer by rounding and places the result in a four-byte integer output vector ivo, for n points.

### **MATHEMATICAL EQUIVALENT:**

$$\text{ivo}[i*\text{inco}] = \text{rv1}[i*\text{incl}], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

Values in rv1 should be between  $-2^{32}$  and  $+2^{32}-1$ .

## **fixbn**

*Vector Fix to One-byte Integer and Round*

### **C USAGE:**

```
float rv1[];
unsigned char bvo[];
long incl, inco, n;
fixbn_(rv1, &incl, bvo, &inco, &n);
-or-
fixbn(rv1, incl, bvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1()
byte bvo()
integer*4 incl, inco, n
call fixbn(rv1, incl, bvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
bvo	(out) - One-byte integer output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points to output

### **SUBROUTINE DESCRIPTION:**

Converts the real input vector rv1 to integer by rounding and places the result in an one-byte integer output vector bvo, for n points.

### **MATHEMATICAL EQUIVALENT:**

$$bvo[i*inco] = rv1[i*incl], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

Values in rv1 should be between 0 and 255.



## **flt2**

## *Float Integer (2 Byte) Vector*

### **C USAGE:**

```
short sv1[];  
float rvo[];  
long incl, inco, n;  
flt2_(sv1, &incl, rvo, &inco, &n);  
-or-  
flt2(sv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
integer*2 sv1()  
real rvo()  
integer*4 incl, inco, n  
call flt2(sv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

sv1	(inp) - Integer input vector
incl	(inp) - Input vector increment
rvo	(out) - Real vector output
inco	(inp) - Output vector increment
n	(inp) - Number of points to float

### **SUBROUTINE DESCRIPTION:**

Converts the input two byte integer vector sv1 to single precision floating point and puts the result in output vector rvo, for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{FLOAT}\{ sv1[i*incl] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **flt2iq**

*Float Integer (2 Byte) I,Q pairs and demux 2 channels*

### **C USAGE:**

```
short sv1[];
float cvo1[], cvo2[];
long incl, n;
flt2iq_(sv1, &incl, cvo1, cvo2, &n);
-or-
flt2iq(sv1, incl, cvo1, cvo2, n);
```

### **FORTRAN USAGE:**

```
integer*2 sv1()
real cvo1(), cvo2()
integer*4 incl, n
call flt2iq(sv1, incl, cvo1, cvo2, n)
```

### **ARGUMENT DESCRIPTIONS:**

sv1	(inp) - Integer input vector
incl	(inp) - Input vector increment
cvo1	(out) - Complex output (1st I,Q channel)
cvo2	(out) - Complex output (2nd I,Q channel)
n	(inp) - Number of complex outputs in cvo1

### **SUBROUTINE DESCRIPTION:**

Special function to float two, multiplexed I,Q channels stored as 16-bit signed integer values. Output consists of two floating-point complex (I,Q) channels.

### **MATHEMATICAL EQUIVALENT:**

```
cvo1[i] = FLOAT{ sv1[i*4*incl], FLOAT{ sv1[i*4*incl+1]}
cvo2[i] = FLOAT{ sv1[i*4*incl+2], FLOAT{ sv1[i*4*incl+3]}
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

Better performance is realized if output vectors are quad-word aligned.

## **flt4**

## *Float Integer (4 Byte) Vector*

### **C USAGE:**

```
long iv1[];
float rvo[];
long incl, inco, n;
flt4_(iv1, &incl, rvo, &inco, &n);
-or
flt4(iv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
integer*4 iv1()
real rvo()
integer*4 incl, inco, n
call flt4(iv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

iv1	(inp) - Integer input vector
incl	(inp) - Input vector increment
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points to convert

### **SUBROUTINE DESCRIPTION:**

Converts the input four byte integer vector iv1 to single precision floating point and puts the result in output vector rvo, for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{FLOAT}\{ iv1[i*incl] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **fltb**

*Float Byte*

### **C USAGE:**

```
char bv1[];  
float rvo[];  
long incl, inco, n;  
fltb_(bv1, &incl, rvo, &inco, &n);  
-or-  
fltb(bv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
byte bv1()  
real rvo()  
integer*4 incl, inco, n  
call fltb(bv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

bv1	(inp) - One-byte integer input vector
incl	(inp) - Input vector increment
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points to convert

### **SUBROUTINE DESCRIPTION:**

Converts the one-byte integer input vector bv1 to single precision floating point and places the result in output vector rvo, for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{FLOAT}\{ bv1[i*incl] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **fltbu**

*Float Byte Unsigned*

### **C USAGE:**

```
char bv1[];  
float rvo[];  
long incl, inco, n;  
fltbu_(bv1, &incl, rvo, &inco, &n);  
-or-  
fltbu(bv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
byte bv1()  
real rvo()  
integer*4 incl, inco, n  
call fltbu(bv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

bv1	(inp) - One-byte integer input vector
incl	(inp) - Input vector increment
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points to convert

### **SUBROUTINE DESCRIPTION:**

Converts the one-byte unsigned integer input vector bv1 to single precision floating point and places the result in vector rvo, for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{FLOAT}\{ bv1[i*incl] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **fxsl2n**

*Vector Scale, Limit, & Fix to Two-byte Integer and Round*

### **C USAGE:**

```
float rv1[];
short svo[];
long incl, inco, n;
float rs1, rs2;
fxsl2n_(rv1, &incl, &rs1, &rs2, svo, &inco, &n);
-or-
fxsl2n(rv1, incl, &rs1, &rs2, svo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1()
integer*2 svo()
integer*4 incl, inco, n
real rs1, rs2
call fxsl2n(rv1, incl, rs1, rs2, svo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rs1	(inp) - Real scalar multiplier
rs2	(inp) - Real scalar adder
svo	(out) - Two-byte integer output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points to output

### **SUBROUTINE DESCRIPTION:**

Converts the real input vector rv1 by multiplying by rs1, adding rs2, limiting to  $2^{15}$  and  $2^{15}-1$  and rounding to nearest integer value. The result is placed in a two-byte output integer vector svo, for n points.

### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    tmp = rv1[i*incl] * rs1 + rs2
    tmp = MAX{ -215, tmp }
    tmp = MIN{ 215- 1, tmp }
    svo[i*inco] = INT{ tmp }
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **fxsl4n**

*Vector Scale, Limit, & Fix to Four-byte Integer & Round*

### **C USAGE:**

```
float rv1[];
long ivo[];
long incl, inco, n;
float rs1, rs2;
fxsl4n_(rv1, &incl, &rs1, &rs2, ivo, &inco, &n);
-or-
fxsl4n(rv1, incl, &rs1, &rs2, ivo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1()
integer*4 ivo()
integer*4 incl, inco, n
real rel, rs2
call fxsl4n(rv1, incl, rs1, rs2, ivo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rs1	(inp) - Real scalar multiplier
rs2	(inp) - Real scalar adder
ivo	(out) - Four-byte integer output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points to output

### **SUBROUTINE DESCRIPTION:**

Converts the real input vector rv1 by multiplying by rs1, adding rs2, limiting that to between  $-2^{31}$  and  $2^{31}-1$ , and rounding to nearest integer value. The result is placed in a four-byte integer output vector ivo, for n points.

### **MATHEMATICAL EQUIVALENT:**

```
tmp = rv1[i*incl] * rs1 + rs2
tmp = MAX{ -231, tmp }
tmp = MIN{ 231 - 1, tmp }
ivo[i*inco] = INT{ tmp }
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **fxslbn**

### *Vector Scale, Limit, & Fix to One-byte Integer & Round*

#### **C USAGE:**

```
float rv1[];
unsigned char bvo[];
long incl, inco, n;
float rs1, rs2;
fxslbn_(rv1, &incl, &rs1, &rs2, bvo, &inco, &n);
-or-
fxslbn(rv1, incl, &rs1, &rs2, bvo, inco, n);
```

#### **FORTRAN USAGE:**

```
real rv1()
byte bvo()
integer*4 incl, inco, n
real rs1, rs2
call fxslbn(rv1, incl, rs1, rs2, bvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rs1	(inp) - Real scalar multiplier
rs2	(inp) - Real scalar adder
bvo	(out) - One-byte integer output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points in output vector

#### **SUBROUTINE DESCRIPTION:**

Converts the real input vector rv1 by multiplying by rs1, adding rs2, limiting between 0 and 255, and rounding to nearest integer value. The result is placed in the one-byte integer output vector bvo, for n points.

#### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    tmp = rv1[i*incl] * rs1 + rs2
    tmp = MAX{ 0, tmp }
    tmp = MIN{ 255, tmp }
    bvo[i*inco] = INT{ tmp }
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **gcosf**

## *Vector Generate Cosines*

### **C USAGE:**

```
float rvo[];  
float rs1, rs2;  
long inco, n,  
gcosf_(rvo, &inco, &rs1, &rs2, &n);  
-or-  
gcosf(rvo, inco, &rs1, &rs2, n);
```

### **FORTRAN USAGE:**

```
real rvo()  
real rs1, rs2  
integer*4 inco, n  
call gcosf(rvo, inco, rs1, rs2, n)
```

### **ARGUMENT DESCRIPTIONS:**

rvo	(out) - Real output vector
inco	(inp) - Output vector increment
rs1	(inp) - Real scalar frequency
rs2	(inp) - Real scalar initial phase
n	(inp) - Number of points in circle

### **SUBROUTINE DESCRIPTION:**

Output vector rvo is filled with cosine values computed from the initial phase rs2 and frequency rs1, for n points. rs1 and rs2 are specified in units of fraction of a circle.

### **MATHEMATICAL EQUIVALENT:**

$$\text{rvo}[i * \text{inco}] = \text{COS}\{ 2\pi * (\text{rs1} * (i-1) + \text{rs2}) \}$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## gcexp

### *Generate Complex Exponential Vector with Constant Rotation*

#### C USAGE:

```
float cvo[][2], float cs1[2], cs2[2];
long inco, n;
gcexp_(cs1, cs2, cvo, &inco, &n);
-or-
gcexp(cs1, cs2, cvo, inco, n);
```

#### FORTRAN USAGE:

```
complex cvo(), cs1, cs2
integer*4 inco, n
call gcexp(cs1, cs2, cvo, inco, n)
```

#### ARGUMENT DESCRIPTIONS:

cs1	(inp) - Complex input scalar (phase step)
cs2	(inp) - Complex input scalar (initial phase)
cvo	(out) - Complex output vector
inco	(inp) - Sample increment of cvo
n	(inp) - Number of samples to output

#### SUBROUTINE DESCRIPTION:

Generates a complex exponential consisting of a unit vector rotating at a constant angular frequency. Each complex output point contains the cosine and sine of the rotation angle in the real and imaginary parts, respectively. Input complex scalar cs1 contains the cosine and sine of the angular increment between output points. Input complex scalar cs2 contains the cosine and sine of the starting phase angle.

For maximum execution speed, each output point is computed from the complex multiplication of the previous point by the complex phase step. For very long vectors (n>10,000) this may introduce cumulative roundoff errors when the phase step is large.

#### MATHEMATICAL EQUIVALENT:

for i=0 to n-1:

$$\begin{aligned} \text{cvo}[i*\text{inco}].\text{r} &= \text{COS}\{a+i*b\} \\ \text{cvo}[i*\text{inco}].\text{i} &= \text{SIN}\{a+i*b\} \end{aligned}$$

where:

$$\begin{aligned} \text{cs1.r} &= \text{COS}\{b\} & \text{cs2.r} &= \text{COS}\{a\} \\ \text{cs1.i} &= \text{SIN}\{b\} & \text{cs2.i} &= \text{SIN}\{a\} \end{aligned}$$

for i=0 to n-1:

$$\begin{aligned} \text{cvo}[i*\text{inco}].\text{r} &= \text{cs1.r}*\text{cvo}[(i-1)*\text{inco}].\text{r} - \text{cs1.i}*\text{cvo}[(i-1)*\text{inco}].\text{i} \\ \text{cvo}[i*\text{inco}].\text{i} &= \text{cs1.r}*\text{cvo}[(i-1)*\text{inco}].\text{i} + \text{cs1.i}*\text{cvo}[(i-1)*\text{inco}].\text{r} \end{aligned}$$

#### RESTRICTIONS & SPECIAL CONDITIONS:

None

## genfilt

### *Generate Complex or Real FIR Filter*

#### C USAGE:

```
float rs1, rs2, rs3, rs4, rs5, rs6, rs7;
double *dvo[];
long flg;

long genfilt_(&rs1, &rs2, &rs3, &rs4, &rs5, &rs6, &rs7, dvo, &flg);
-or-
long genfilt(rs1, rs2, rs3, rs4, rs5, rs6, &rs7, dvo, flg);
```

#### FORTTRAN USAGE:

```
real rs1, rs2, rs3, rs4, rs5, rs6, rs7
real*8 dvo()
integer*4 flg
call genfilt(rs1, rs2, rs3, rs4, rs5, rs6, rs7, dvo, flg)
```

#### ARGUMENT DESCRIPTIONS:

rs1	(inp) - Real Sample Rate
rs2	(inp) - Real Center Frequency
rs3	(inp) - Real Bandwidth
rs4	(inp) - Real Transition Width
rs5	(inp) - Real Passband Ripple
rs6	(inp) - Real Stop Ripple
rs7	(out) - Real Number of coefficients
dvo	(out) - Generated Filter Coefficients
flg	(inp) - Filter type 0=Real Filter, 1=Complex Filter

#### SUBROUTINE DESCRIPTION:

Given the parameters, genfilt generates a Parks-McClean FIR filter using the remez algorithm.

#### MATHEMATICAL EQUIVALENT:

#### RESTRICTIONS & SPECIAL CONDITIONS:

This routine allocates the necessary memory for the filter on the heap using realloc, multiple calls to the routine using the same pointer is okay, however when the user is done with the pointer to the filter, it is the users responsibility to call free, to release the memory.

## genorder

*Generate Complex or Real FIR Filter Order*

### C USAGE:

```
float rs1, rs2, rs3, rs4, rs5, rs6, rs7;  
double *dvo[];  
long flg;  
long genorder_(&rs1, &rs2, &rs3, &rs4, &rs5, &rs6, &rs7, flg);  
-or-  
long genorder(rs1, rs2, rs3, rs4, rs5, rs6, &rs7, flg);
```

### FORTRAN USAGE:

```
real rs1, rs2, rs3, rs4, rs5, rs6, rs7  
real*8 dvo()  
integer*4 flg  
call genorder(rs1, rs2, rs3, rs4, rs5, rs6, rs7, flg)
```

### ARGUMENT DESCRIPTIONS:

rs1	(inp) - Real Sample Rate
rs2	(inp) - Real Center Frequency
rs3	(inp) - Real Bandwidth
rs4	(inp) - Real Transition Width
rs5	(inp) - Real Passband Ripple
rs6	(inp) - Real Stop Ripple
rs7	(out) - Real Number of coefficients
flg	(inp) - Filter type 0=Real Filter, 1=Complex Filter

### SUBROUTINE DESCRIPTION:

Given the parameters, genorder computes the number of coefficients that will be needed to generate a Parks-McClearn FIR filter using genfilt.

### MATHEMATICAL EQUIVALENT:

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## hamm

*Apply a Hamming Window to a Real Vector*

### C USAGE:

```
float rv1[], rvo[];
long incl, inco, n;
hamm_(rv1, &incl, rvo, &inco, &n);
-or-
hamm(rv1, incl, rvo, inco, n);
```

### FORTRAN USAGE:

```
real rv1(), rvo()
integer*4 incl, inco, n
call hamm(rv1, incl, rvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process

### SUBROUTINE DESCRIPTION:

Multiplies each element of the input vector rv1 by the Hamming window.

### MATHEMATICAL EQUIVALENT:

for j=0 to n-1:  
$$rvo[j*inco] = W * rv1[j * incl] * (0.54 - 0.46 * \cos\{ 2\pi*j/n \})$$

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## **hann**

*Apply a Hanning Window to a Real Vector*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n, flg;
hann_(rv1, &incl, rvo, &inco, &n, &flg);
-or-
hann(rv1, incl, rvo, inco, n, flg);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n, flg
call hann(rv1, incl, rvo, inco, n, flg)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process
flg	(inp) - Flag
	flg = 0 => weight using .5
	flg = 1 => weight using .8165 (sqrt(2/3))

### **SUBROUTINE DESCRIPTION:**

Multiplies each element of the input vector rv1 by the Hanning window.

### **MATHEMATICAL EQUIVALENT:**

```
for j=0 to n-1:
    rvo[j*inco] = W * rv1[j *incl] * (1.0-COS{ 2π*j/n })
    W = .5 for flg=0
    W = sqrt(2/3) for flg=1
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## hlbrt

## Hilbert Transform

### C USAGE:

```
float rv1[], rvo[];
long n;
hlbrt_(rv1, rvo, &n);
-or-
hlbrt(rv1, rvo, n);
```

### FORTRAN USAGE:

```
real rv1(), rvo()
integer*4 n
call hlbrt(rv1, rvo, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector
rvo	(out) - Real output vector
n	(inp) - Number of samples to process

### SUBROUTINE DESCRIPTION:

Performs an out-of-place Hilbert Transform on a real compact vector by taking a forward FFT, multiplying the complex result by  $-i$ , and taking the inverse FFT. In the algorithm given below, ctemp is a temporary complex vector. The Nyquist point is discarded during the calculation.

### MATHEMATICAL EQUIVALENT:

```
ctemp = real forward FFT { rv1 }
for i=0 to n/2-1:
    temp = IMAG{ ctemp[i] }
    ctemp[i].Im = - REAL { ctemp[i] }
    ctemp[i].Re = temp
ctemp[0] = (0.0, 0.0)
```

### RESTRICTIONS & SPECIAL CONDITIONS:

- 1) Vector rv1 is not overwritten unless it is the same as rvo (which is permissible).
- 2) All vectors must be compact
- 3)  $n=2^M$ ,  $5 \leq M \leq 11$ .

## leveldet

## *Time Series Energy Detector*

### C USAGE:

```
float rv1[], avg, siglevel;;
long noise, gap, sig, thresh, maxsearch, det, n;
leveldet_(rv1, &noise, &gap, &sig, &thresh, &maxsearch, &det, &avg,
&siglevel, &n);      -or-
leveldet(rv1, noise, gap, sig, thresh, maxsearch, &det, &avg, &siglevel,
n);
```

### FORTRAN USAGE:

```
real rv1(), avg, siglevel;
integer*4 noise, gap, sig, thresh, maxsearch, det, n;
call leveldet(rv1, noise, gap, sig, thresh, maxsearch, &det, &avg,
&siglevel, n);
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector
noise	(inp) - Size of noise window (samples)
gap	(inp) - Size of gap between noise window and signal window (samples)
sig	(inp) - Size of signal window (samples)
thresh	(inp) - Threshold level for detection (linear)
maxsearch	(inp) - Maximum number of samples to continue search after initial detection
det	(out) - Index of sample where detection occurred (if -1, no detection found)
avg	(out) - decay average of noise (linear)
siglevel	(out) - sum of energy in signal window at time of detection
n	(inp) - Number of points to process

### SUBROUTINE DESCRIPTION:

leveldetect performs a time-domain search for pulse or edge that is greater than the background noise. The method used is slide windows through the data that are computed for each point in the data. There are 2 types of windows, signal and noise. The noise window is the leftmost window, which is seperated by the gap width specified in "gap" from the signal window, which is seperated by "gap" from the rightmost window, another noise window. Such that a window configuration a noise window of 5, a gap of 3, and a signal window of 2 would look like this: NNNNNGGGSSGGGNNNN.

These windows are moved through the data one sample at a time. The energy in the noise windows is averaged to compute the background noise. The energy in the signal window is averaged to compute signal level. The dection in is done in two phases: in Phase I if the energy in the signal windows is at least thresh times greater than the signal window a detection has been found. The second phase is to search from the point of the initial detection for a duration of maxsearch to see if any of the samples are a minim of 5% greater in magnitude than the original detection, thus selecting the greatest amplitude for the detection. This is used so there is some "parameter insensitivity".

When a detection is found the zero based index is returned in det (ie. the first sample is 0). If no detection is found a -1 is returned in det.

### MATHEMATICAL EQUIVALENT:

N/A



**RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Only one detection is returned at a time, to process an entire block of data, leveledetect must be called multiple times with the appropriate adjustments for the starting point and number of samples to process.
- 2)  $n < \text{size of rv1} - \text{ROUND}\{\text{sig}/2.0\} + \text{gap} + \text{noise}$

## **lveq**

*Logical Vector Equal*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, n;
lveq_(rv1, &incl, rv2, &inc2, rvo, &inco, &n);
-or-
lveq(rv1, incl, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, inco, n
call lveq(rv1, incl, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment 1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment 2
rvo	(out) - Real vector output
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Compare n points in two input vector points: if point in input vector rv1 is equal to point in input vector rv2, put 1.0 into output vector rvo; else put 0.0 into output vector rvo.

### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    if rv1[i*incl] = rv2[i*inc2] then rvo[i*inco] = 1
    else rvo[i*inco] = 0
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## lvge

*Logical Vector Greater Than or Equal*

### C USAGE:

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, n;
lvge_(rv1, &incl, rv2, &inc2, rvo, &inco, &n);
-or-
lvge(rv1, incl, rv2, inc2, rvo, inco, n);
```

### FORTRAN USAGE:

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, inco, n
call lvge(rv1, incl, rv2, inc2, rvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment 1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment 2
rvo	(out) - Real vector output
inco	(out) - Output vector increment
n	(inp) - Number of samples to process

### SUBROUTINE DESCRIPTION:

Compare n points in two input vector points: if point in input vector rv1 is greater than or equal to point in input vector rv2, put 1.0 into output vector rvo; else put 0.0 into output vector.

### MATHEMATICAL EQUIVALENT:

```
for i=0 to n-1
    if rv1 [i*incl] >=rv2[i*inc2] then rvo[i*inco] = 1
    else rvo[i*inco] = 0
```

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## lvgt

*Logical Vector Greater*

### C USAGE:

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, n;
lvgt_(rv1, &incl, rv2, &inc2, rvo, &inco, &n);
-or-
lvgt(rv1, incl, rv2, inc2, rvo, inco, n);
```

### FORTRAN USAGE:

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, inco, n
call lvgt(rv1, incl, rv2, inc2, rvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment 1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment 2
rvo	(out) - Real vector output
inco	(out) - Vector increment
n	(inp) - Number of samples to process

### SUBROUTINE DESCRIPTION:

Compare n points in two input vector points: if point in input vector rv1 is greater than the point in input vector rv2, put 1.0 into output vector rvo; else put 0.0 into output vector.

### MATHEMATICAL EQUIVALENT:

```
for i=0 to n-1:
    if rv1[i*incl] > rv2[i*inc2] then rvo[i*inco] = 1.0
    else rvo[i*inco] = 0.0
```

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## lvle

*Logical Vector Less Than or Equal*

### C USAGE:

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, n;
lvle_(rv1, &incl, rv2, &inc2, rvo, &inco, &n);
-or-
lvle(rv1, incl, rv2, inc2, rvo, inco, n);
```

### FORTRAN USAGE:

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, inco, n call
lvle(rv1, incl, rv2, inc2, rvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment 1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment 2
rvo	(out) - Real vector output
inco	(out) - Vector increment for rvo
n	(inp) - Number of samples to process

### SUBROUTINE DESCRIPTION:

Compare n points in two input vector points: if point in input vector rv1 is less than or equal to point in input vector rv2, put 1.0 into output vector rvo; else put 0.0 into output vector.

### MATHEMATICAL EQUIVALENT:

```
for i=0 to n-1:
    if rv1 [i*incl] <= rv2[i*inc2] then rvo[i*inco] = 1.0
    else rvo[i*inco] = 0.0
```

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## lvlt

## *Logical Vector Less*

### C USAGE:

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, n;
lvlt_(rv1, &incl, rv2, &inc2, rvo, &inco, &n);
-or-
lvlt(rv1, incl, rv2, inc2, rvo, inco, n);
```

### FORTRAN USAGE:

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, inco, n
call lvlt(rv1, incl, rv2, inc2, rvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment 1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment 2
rvo	(out) - Real vector output
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process

### SUBROUTINE DESCRIPTION:

Compare n points in two input vectors: if point in input vector rv1 is less than the point in input vector rv2, put 1.0 into output vector rvo; else put 0.0 into output vector.

### MATHEMATICAL EQUIVALENT:

```
for i=0 to n-1
    if rv1 [i*incl] < rv2[i*inc2] then rvo[i*inco] = 1.0
    else rvo[i*inco] = 0.0
```

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## lvne

*Logical Vector Not Equal*

### C USAGE:

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, n;
lvne_(rv1, &incl, rv2, &inc2, rvo, &inco, &n);
-or-
lvne(rv1, incl, rv2, inc2, rvo, inco, n);
```

### FORTRAN USAGE:

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, inco, n
call lvne(rv1, incl, rv2, inc2, rvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment 1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment 2
rvo	(out) - Real vector output
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process

### SUBROUTINE DESCRIPTION:

Compare n points in two input vectors: if point on input vector rv1 is not equal to point in input vector rv2, put 1.0 into output vector rvo; else put 0.0 into output vector.

### MATHEMATICAL EQUIVALENT:

```
for i=0 to n-1:
    if rv1[i*incl]<= rv2[i*inc2] then rvo[i*inco] = 1.0
    else rvo[i*inco] = 0.0
```

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## lvnot

*Logical Vector Not Equal to Zero*

### C USAGE:

```
float rv1[], rvo[];
long incl, inco, n;
lvnot_(rv1, &incl, rvo, &inco, &n);
-or-
lvnot(rv1, incl, rvo, inco, n);
```

### FORTRAN USAGE:

```
real rv1(), rvo()
integer*4 incl, inco, n
call lvnot(rv1, incl, rvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector
ind	(inp) - Vector increment for rv1
rvo	(out) - Real vector output
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process

### SUBROUTINE DESCRIPTION:

Compare n points in two input vectors: if point in input vector rv1 is not equal to zero, put 1.0 into output vector rvo; else put 0.0 into output vector.

### MATHEMATICAL EQUIVALENT:

```
for i=0 to n-1:
    if rv1[i*inc 1] <> 0 then rvo[i*inco] = 1.0
    else rvo[i*inco] = 0.0
```

### RESTRICTIONS & SPECIAL CONDITIONS:

None



## **maxmgv**

## *Maximum Magnitude Element of Vector*

### **C USAGE:**

```
float rv1[], rso;
long incl, idx, n;
maxmgv_(rv1, &incl, &rso, &idx, &n);
-or-
maxmgv(rv1, incl, &rso, &idx, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso
integer*4 incl, idx, n
call maxmgv(rv1, incl, rso, idx, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rso	(out) - Maximum magnitude output
idx	(inp) - Index of maximum magnitude
n	(inp) - Number of points to process

### **SUBROUTINE DESCRIPTION:**

The vector rv1 is searched from the beginning for the element with the greatest magnitude. The position of the found element is returned in idx, but is not a traditional offset. It is the position within the examined elements only and is dependent on the increment used. The first position is 0.

### **MATHEMATICAL EQUIVALENT:**

```
i=0
rso = |rv1[0]|
while i < n:
    if |rv1[i]| > rso then
        rso = |rv1[i]|
        idx = i/incl
        i = i + incl
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **maxv**

*Find the Maximum Value and its Location*

### **C USAGE:**

```
float rv1[], rso;
long incl, idx, n;
maxv_(rv1, &incl, &rso, &idx, &n);
-or-
maxv(rv1, incl, &rso, &idx, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso
integer*4 incl, idx, n
call maxv(rv1, incl, rso, idx, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rso	(out) - Real scalar output
idx	(out) - Index of maximum point
n	(inp) - Number of points to process

### **SUBROUTINE DESCRIPTION:**

The vector rv1 is searched from the beginning for the element with the highest value. The position of the found element is returned in idx, but is not a traditional offset. It is the position within the examined elements only and is dependent on the increment used. The first position is 0.

### **MATHEMATICAL EQUIVALENT:**

```
i=0
rso = rv1[0]
while i < n:
    if rv1[i] > rso then
        rso = rv1[i]
        idx = i/incl
        i = i + incl
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **meamgv**

*Mean Magnitude of Real Vector*

### **C USAGE:**

```
float rv1[], rso;
long incl, n;
meamgv_(rv1, &incl, &rso, &n);
-or-
meamgv(rv1, incl, &rso, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso
integer*4 incl, n
call meamgv(rv1, incl, rso, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rso	(out) - Real scalar output
n	(inp) - Number of points to process

### **SUBROUTINE DESCRIPTION:**

Mean value of absolute value of elements of rv1 is returned in real scalar rso.

### **MATHEMATICAL EQUIVALENT:**

$$rso = \text{SUM} \{ |rv1[i*incl]| \} / n, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **meanv**

*Mean of Real Vector*

### **C USAGE:**

```
float rv1[], rso;  
long incl, n;  
meanv_(rv1, &incl, &rso, &n);  
-or-  
meanv(rv1, incl, &rso, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso  
integer*4 incl, n  
call meanv(rv1, incl, rso, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rso	(out) - Real scalar output
n	(inp) - Number of points to process

### **SUBROUTINE DESCRIPTION:**

Mean value of elements in rv1 is returned in real scalar rso.

### **MATHEMATICAL EQUIVALENT:**

$$rso = \text{SUM} \{ rv1[i*incl] \} / n, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **measqv**

*Mean of Square of Real Vector*

### **C USAGE:**

```
float rv1[], rso;  
long incl, n;  
measqv_(rv1, &incl, &rso, &n);  
-or-  
measqv(rv1, incl, &rso, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso  
integer*4 incl, n  
call measqv(rv1, incl, rso, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rso	(out) - Real scalar output
n	(inp) - Number of points to process

### **SUBROUTINE DESCRIPTION:**

Mean value of square of elements of rv1 is returned in real scalar rso.

### **MATHEMATICAL EQUIVALENT:**

$$rso = \text{SUM} \{ rv1[i*incl]^2 \} / n, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **minmgv**

*Find the Minimum Magnitude and its Location*

### **C USAGE:**

```
float rv1[], rso;
long incl, idx, n;
minmgv_(rv1, &incl, &rso, &idx, &n);
-or-
minmgv(rv1, incl, &rso, &idx, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso
integer*4 incl, idx, n
call minmgv(rv1, incl, rso, idx, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rso	(out) - Real scalar output
idx	(out) - Position in vector
n	(inp) - Number of points to process

### **SUBROUTINE DESCRIPTION:**

The vector rv1 is searched from the beginning for the element with the least magnitude. The position of the found element is returned in idx, but is not a traditional offset. It is the position within the examined elements only and is dependent on the increment used. The first position is 0.

### **MATHEMATICAL EQUIVALENT:**

```
i=0
rso = |rv1[0]|
while i < n:
    if |rv1[i]| < rso then
        rso = |rv1[i]|
        idx = i/incl
        i = i + incl
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **minv**

*Find the Minimum Value and its Position*

### **C USAGE:**

```
float rv1[], rso;  
long incl, idx, n;  
minv_(rv1, &incl, &rso, &idx, &n);  
-or-  
minv(rv1, incl, &rso, &idx, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso  
integer*4 incl, idx, n  
call minv(rv1, incl, rso, idx, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rso	(out) - Real scalar output
idx	(out) - Position of minimum
n	(inp) - Number of points in output vector

### **SUBROUTINE DESCRIPTION:**

The vector rv1 is searched from the beginning for the element with the least value. The position of the found element is returned in idx, but is not a traditional offset. It is the position within the examined elements only and is dependent on the increment used. The first position is 0.

### **MATHEMATICAL EQUIVALENT:**

```
i=0  
rso = rv1[0]  
while i < n:  
    if rv1[i] < rso then  
        rso = rv1[i]  
        idx = i/incl  
        i = i + incl
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **mvessq**

*Mean of Signed Square of Vector Elements*

### **C USAGE:**

```
float rv1[], rso;  
long incl, n;  
mvessq_(rv1, &incl, &rso, &n);  
-or-  
mvessq(rv1, incl, &rso, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso  
integer*4 incl, n  
call mvessq(rv1, incl, rso, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rso	(out) - Real scalar output
n	(inp) - Number of points in output vector

### **SUBROUTINE DESCRIPTION:**

Returns the mean value of the signed squares of the elements of rv1.

### **MATHEMATICAL EQUIVALENT:**

$$rso = \text{SUM} \{ rv1[i*incl] * |rv1[i*incl]| \} / n, i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **mxmla**

## *Real Matrix plus Matrix Product*

### **C USAGE:**

```
float rm1[], rm2[], rmo[];
long incc1, incr1, incc2, incr2, incco, incro, nco, nro, ncr;
mxmla_(rm1, &incc1, &incr1, rm2, &incc2, &incr2, rmo, &incco, &incro,
&nro, &nco, &ncr);
-or-
mxmla(rm1, incc1, incr1, rm2, incc2, incr2, rmo, incco, incro, nro, nco,
ncr);
```

### **FORTRAN USAGE:**

```
real rm1(), rm2(), rmo()
integer*4 incc1, incr1, incc2, incr2, incco, incro, nco, nro, ncr
call mxmla(rm1, incc1, incr1, rm2, incc2, incr2, rmo, incco, incro, nro,
nco, ncr)
```

### **ARGUMENT DESCRIPTIONS:**

rm1	(inp) - First real matrix
incc1	(inp) - Sample increment between columns
incr1	(inp) - Sample increment between rows
rm2	(inp) - Second real matrix to move
incc2	(inp) - Sample increment between columns
incr2	(inp) - Sample increment between rows
rmo	(i/o) - Real output matrix
incco	(inp) - Sample increment between columns
incro	(inp) - Sample increment between rows
nro	(inp) - Number of rows in output matrix
nco	(inp) - Number of columns in output matrix
ncr	(inp) - Number of columns in rm1 and rows in rm2

### **SUBROUTINE DESCRIPTION:**

Increases the value of matrix rmo by the product of the two matrices rm1 and rm2

### **MATHEMATICAL EQUIVALENT:**

$$rmo = rmo + (rm1 * rm2) \text{ (elements set depend on increments)}$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## mxm1s

## *Real Matrix minus Matrix Product*

### C USAGE:

```
float rm1[], rm2[], rmo[];
long incc1, incrl, incc2, incr2, incco, incro, nco, nro, ncr;
mxm1s_(rm1, &incc1, &incrl, rm2, &incc2, &incr2, rmo, &incco, &incro,
&nro, &nco, &ncr);
-or-
mxm1s(rm1, incc1, incrl, rm2, incc2, incr2, rmo, incco, incro, nro, nco,
ncr);
```

### FORTRAN USAGE:

```
real rm1(), rm2(), rmo()
integer*4 incc1, incrl, incc2, incr2, incco, incro, nco, nro, ncr
call mxm1s(rm1, incc1, incrl, rm2, incc2, incr2, rmo, incco, incro, nro,
nco, ncr)
```

### ARGUMENT DESCRIPTIONS:

rm1	(inp) - First real matrix
incc1	(inp) - Sample increment between columns
incrl	(inp) - Sample increment between rows
rm2	(inp) - Second real matrix to move
incc2	(inp) - Sample increment between columns
incr2	(inp) - Sample increment between rows
rmo	(i/o) - Real output matrix
incco	(inp) - Sample increment between columns
incro	(inp) - Sample increment between rows
nro	(inp) - Number of rows in output matrix
nco	(inp) - Number of columns in output matrix
ncr	(inp) - Number of columns in rm1 and rows in rm2

### SUBROUTINE DESCRIPTION:

Decreases the value of matrix rmo by the product of the two matrices rm1 and rm2

### MATHEMATICAL EQUIVALENT:

$$rmo = rmo - (rm1 * rm2) \text{ (elements set depend on increments)}$$

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## **mxmov**

## *Real Matrix Move*

### **C USAGE:**

```
float rml[], rmo[];
long inccl, incrl, incco, incro, nco, nro;
mxmov_(rml, &inccl, &incrl, rmo, &incco, &incro, &nro, &nco);
-or-
mxmov(rml, inccl, incrl, rmo, incco, incro, nro, nco);
```

### **FORTRAN USAGE:**

```
real rml(), rmo()
integer*4 inccl, incrl, incco, incro, nco, nro
call mxmov(rml, inccl, incrl, rmo, incco, incro, nro, nco)
```

### **ARGUMENT DESCRIPTIONS:**

rml	(inp) - First real matrix
inccl	(inp) - Sample increment between columns
incrl	(inp) - Sample increment between rows
rmo	(out) - Real output matrix
incco	(inp) - Sample increment between columns
incro	(inp) - Sample increment between rows
nro	(inp) - Number of rows in output matrix
nco	(inp) - Number of columns in output matrix

### **SUBROUTINE DESCRIPTION:**

This routine moves real elements from one strided matrix to a second strided matrix.

### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to nco-1:
    for j=0 to nro-1:
        rmo[i*incro+j*incco]=rml[i*incrl+j*inccl];
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

It is the intent of this routine to provide a means of manipulating sub-matrices and executing various symmetry operations such as transpose. For simply moving a compact matrix, vmov will provide faster execution.

## mxmul

## *Real Matrix Multiply*

### C USAGE:

```
float rm1[], rm2[], rmo[];
long incc1, incr1, incc2, incr2, incco, incro, nco, nro, ncr;
mxmul_(rm1, &incc1, &incr1, rm2, &incc2, &incr2, rmo, &incco, &incro,
&nro, &nco, &ncr);
-or-
mxmul(rm1, incc1, incr1, rm2, incc2, incr2, rmo, incco, incro, nro, nco,
ncr);
```

### FORTRAN USAGE:

```
real rm1(), rm2(), rmo()
integer*4 incc1, incr1, incc2, incr2, incco, incro, nco, nro, ncr
call mxmul(rm1, incc1, incr1, rm2, incc2, incr2, rmo, incco, incro, nro,
nco, ncr)
```

### ARGUMENT DESCRIPTIONS:

rm1	(inp) - First real matrix
incc1	(inp) - Sample increment between columns
incr1	(inp) - Sample increment between rows
rm2	(inp) - Second real matrix to move
incc2	(inp) - Sample increment between columns
incr2	(inp) - Sample increment between rows
rmo	(out) - Real output matrix
incco	(inp) - Sample increment between columns
incro	(inp) - Sample increment between rows
nro	(inp) - Number of rows in output matrix
nco	(inp) - Number of columns in output matrix
ncr	(inp) - Number of columns in rm1 and rows in rm2

### SUBROUTINE DESCRIPTION:

Performs a real matrix multiply.

### MATHEMATICAL EQUIVALENT:

$$rmo = rm1 * rm2 \text{ (elements set depend on increments)}$$

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## **polar**

### *Rectangular to Polar Coordinate Conversion*

#### **C USAGE:**

```
float cv1[][2], cvo[][2];
long incl, inco, n;
polar_(cv1, &incl, cvo, &inco, &n);
-or-
polar(cv1, incl, cvo, inco, n);
```

#### **FORTRAN USAGE:**

```
complex cv1(), cvo()
integer*4 incl, inco, n
call polar(cv1, incl, cvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Input vector increment
cvo	(out) - Complex output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points in Output vector

#### **SUBROUTINE DESCRIPTION:**

The polar coordinates of each rectangular input pair are determined and written to the output vector cvo. For cvo, the real part is the radius and the imaginary part is the angle in radians.

#### **MATHEMATICAL EQUIVALENT:**

for i=0 to n-1:

$$\text{cvo}[i*\text{inco}].r = \text{SQRT}\{ \text{Re}\{ \text{cv1}[i*\text{incl}] \}^2 + (\text{Im}\{ \text{cv1}[i*\text{incl}] \})^2 \}$$
$$\text{cvo}[i*\text{inco}].i = \text{ARCTAN}\{ \text{Im}\{ \text{cv1}[i*\text{incl}] \} / \text{Re}\{ \text{cv1}[i*\text{incl}] \} \}$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **rect**

### *Polar to Rectangular Coordinate Conversion*

#### **C USAGE:**

```
float cv1[][2], cvo[][2];
long incl, inco, n;
rect_(cv1, &incl, cvo, &inco, &n);
-or-
rect(cv1, incl, cvo, inco, n);
```

#### **FORTRAN USAGE:**

```
complex cv1(), cvo()
integer*4 incl, inco, n
call rect(cv1, incl, cvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Input vector increment
cvo	(out) - Complex output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points in output vector

#### **SUBROUTINE DESCRIPTION:**

The rectangular coordinates are calculated from the polar coordinate pair in complex input vector cv1 and stored in complex vector cvo. For the input vector cv1, the real part is the radius and the imaginary part is the angle in radians.

#### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    cvo[i*inco].r = Re{ cv1[i*incl] } * COS{ Im{ cv1[i*incl] } }
    cvo[i*inco].i = Re{ cv1[i*incl] } * SIN{ Im{ cv1[i*incl] } }
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## reqs

*Find Location of First Element Equal to a Scalar*

### C USAGE:

```
float rv1[], rs1;  
long incl, idx, n;  
reqs_(rv1, &incl, &rs1, &n, &idx);  
-or-  
reqs(rv1, incl, &rs1, n, &idx);
```

### FORTRAN USAGE:

```
real rv1(), rs1  
integer*4 incl, idx, n  
call reqs(rv1, incl, rs1, n, idx)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rs1	(inp) - Real scalar input
n	(inp) - Number of points in output vector
idx	(out) - Position of scalar found in rv1

### SUBROUTINE DESCRIPTION:

The vector rv1 is searched from the beginning for an element equal to the scalar rs1. The position of the found element is returned in idx, but is not a traditional offset. It is the position within the examined elements only and is dependent on the increment used. The first position is 0. If no element is found meeting the criteria, -1 is returned for idx.

### MATHEMATICAL EQUIVALENT:

```
i=0  
idx=0  
while i < n:  
    if rv1[i] = rs1 then break  
    i = i + incl  
    idx = idx + 1
```

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## **rfft**

*Real FFT, in Place - Fwd and Inv - Sizes to 2048 Reals*

### **C USAGE:**

```
float cvio[];
long n, flg;
rfft_(cvio, &n, &flg);
-or-
rfft(cvio, n, flg);
```

### **FORTRAN USAGE:**

```
complex cvio()
integer*4 n, flg
call rfft(cvio, n, flg)
```

### **ARGUMENT DESCRIPTIONS:**

cvio	(inp) - Real/complex input vector
n	(inp) - FFT size (number of real points)
flg	(inp) - Flag selecting transform direction +1 => forward, -1 => inverse

### **SUBROUTINE DESCRIPTION:**

The forward real-to-complex or inverse complex-to-real FFT of the input vector is computed in place.

### **MATHEMATICAL EQUIVALENT:**

for  $k=0, m-1$   
 $Y(m) = \text{SUM } \{x[k] \cdot W^{m \cdot k}\}$ , for  $m=0$  to  $n-1$   
where  $x = \text{cvio}$ ,  $Y = \text{cvio}$ ,  $j = \text{SQRT}\{-1\}$   
 $W = \exp(-j \cdot 2\pi/n)$  for  $\text{flg} = +1$   
 $W = \exp(+j \cdot 2\pi/n)$  for  $\text{flg} = -1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Size  $n$  must be a power of 2 in the range  $16 \leq n \leq 2048$ .
- 2) Input  $cv$  must be a compact vector (stride = 2), aligned on an 8-byte boundary.
- 3) `fftwts` must be called before calling `rfft`, with size  $\geq$  FFT size/2.
- 4) Forward transform places the real part of the Nyquist point  $cv[n/2]$  in the imaginary part of  $cv[0]$ .
- 5) Inverse transform assumes the real part of the Nyquist point  $cv[n/2]$  to be in the imaginary part of  $cv[0]$ .



## **rfftf**

*Real Forward FFT - Sizes to 2097152 (2 Meg) Reals*

### **C USAGE:**

```
float rv1[], cvo[][2], rs1;  
long incl, inco, n;  
rfftf_(rv1, &incl, cvo, &inco, &rs1, &n);  
-or  
rfftf(rv1, incl, cvo, inco, &rs1, n);
```

### **FORTRAN USAGE:**

```
complex cvo()  
real rv1(), rs1  
integer*4 incl, inco, n  
call rfftf(rv1, incl, cvo, inco, rs1, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
cvo	(out) - Complex output vector
inco	(inp) - Output vector increment
rs1	(out) - Nyquist value
n	(inp) - FFT size (number of real points)

### **SUBROUTINE DESCRIPTION:**

The forward real-to-complex FFT of the input vector is computed and stored in the output vector.

### **MATHEMATICAL EQUIVALENT:**

for  $k=0, m-1$   
 $Y(m) = \text{SUM} \{ x[k] * [W^{**}(m*k)] \}$ , for  $m=0$  to  $n-1$   
where  $x = rv1$ ,  $Y = cvo$ ,  $j = \text{SQRT} \{ -1 \}$ ,  $W = \text{EXP} \{ -j*2\pi/n \}$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Size  $n$  must be a power of 2 in the range  $16 \leq n \leq 2097152$ .
- 2) Input and output vectors  $rv1$  &  $cvo$  must be aligned on 8-byte boundaries.
- 3) For sizes  $> 2048$ , input and output vectors  $rv1$  &  $cvo$  cannot overlap.
- 4) For sizes  $> 2048$ , input vector  $rv1$  must be compact.
- 5) `fftwts` must be called before calling `rfftf`, with size  $\geq$  FFT size/2.

## **rfffti**

*Real Inverse FFT - Sizes to 2097152 (2 Meg) Reals*

### **C USAGE:**

```
float cv1[][2], rvo[], rs1;  
long incl, inco, n;  
rfffti_(cv1, &incl, &rs1, rvo, &inco, &n);  
-or-  
rfffti(cv1, incl, &rs1, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1()  
real rvo(), rs1  
integer*4 incl, inco, n  
call rfffti(cv1, incl, rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Input vector increment
rs1	(inp) - Nyquist point (real part of cv1 [n/2])
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - FFT size (number of real output points)

### **SUBROUTINE DESCRIPTION:**

The inverse complex-to-real FFT of the input vector is computed and stored in the output vector.

### **MATHEMATICAL EQUIVALENT:**

for k=0,m-1  
 $Y(m) = \text{SUM} \{ x[k] * [W^{**}(m*k)] \}$ , for m=0 to n-1  
where x=cv1, Y = rvo, j = SQRT{ -1 }, W = EXP{ +j\*2 $\pi$ /n }

### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Size n must be a power of 2 in the range  $16 \leq n \leq 2097152$ .
- 2) Input vector cv1 must have an even sample increment, and be aligned on 8-byte boundary.
- 3) Output vector rvo must be aligned on an 8-byte boundary.
- 4) Output vector rvo must be compact (inco must be 1).
- 5) fftwts must be called before calling rfffti, with size  $\geq$  FFT size/2.
- 6) For sizes  $> 2048$ , input vector cv1 is overwritten.
- 7) The pointer to the Nyquist point (real part of cvo[n/2]) may be set to point to the imaginary part of cvo[0].

## **rfftsc**

## *Real FFT Scale and Format*

### **C USAGE:**

```
float cvio[][2];
long n, flg1, flg2;
rfftsc_(cvio, &n, &flg1, &flg2);
-or-
rfftsc(cvio, n, flg1, flg2);
```

### **FORTRAN USAGE:**

```
real cvio()
integer*4 n, flg1, flg2
call rfftsc(cvio, n, flg1, flg2)
```

### **ARGUMENT DESCRIPTIONS:**

cvio	(i/o) - Complex input/output vector
n	(inp) - Number of real points in cvio (n/2 complex)
flg1	(inp) - Format flag
flg2	(inp) - Scale flag

### **SUBROUTINE DESCRIPTION:**

Scales and formats real FFT input/output vector, converting between pure complex form (n/2+1 complex points) and a packed RFFT form (n/2 complex points, with real part of cvio[n/2] packed into the imaginary part of cvio[0]).

### **MATHEMATICAL EQUIVALENT:**

<u>Format flag flg1</u>	<u>Output format</u>
-1,0,1	cvio = cvio
2	cvio[0].im = 0.0
-2	cvio[0].im = 0.0
3	cvio[n/2+1].re = cvio[0].im
	cvio[n/2+1].im = cvio[0].im = 0.0
-3	cvio[0].im = cvio[n/2+1].re

<u>Scale flag flg2</u>	<u>Output scale</u>
0	cvio = cvio
1	cvio = cvio*1/(n*2)
-1	cvio = cvio*1/(n*4)

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **rft2fc**

### *Two Dimensional Real Forward FFT - Column Compact*

#### **C USAGE:**

```
float rv1[][2], cvn[2], cvo[][2];
long inc2, nr, nc;
rft2fc_(rv1, cvn, &inc2, cvo, &nr, &nc);
-or-
rft2fc(rv1, cvn, inc2, cvo, nr, nc);
```

#### **FORTRAN USAGE:**

```
real rv1()
complex cvn(), cvo()
integer*4 inc2, nr, nc
call rft2fc(rv1, cvn, inc2, cvo, nr, nc)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
cvn	(out) - Complex Nyquist vector
inc2	(inp) - Nyquist vector increment
cvo	(out) - Complex output vector
nr	(inp) - Number of rows in rv1
nc	(inp) - Number of columns in rv1

#### **SUBROUTINE DESCRIPTION:**

The two dimensional forward real FFT of the input vector is computed and stored in the output vector. The columns are assumed to be compact.

The two dimensional real FFT is defined as a series of forward real-to-complex FFTs down the columns, followed by forward complex-to-complex FFTs across the rows.

The real FFTs each produce a single complex Nyquist frequency output, which is stored in cvn. A final complex-to-complex FFT of cvn is computed.

#### **MATHEMATICAL EQUIVALENT:**

```
for i = 0 to nc-1:
    fftf(rv1, 1, cvo, 2, cvn[i*inc2], nr)
for i = 0 to nr/2-1:
    fftf(cvo, nr, cvo, nr, nc)
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Input data is assumed to be in column order (FORTRAN)
- 2) Size is restricted to 2048 x 1024 (nr x nc).

## **rft2fr**

### *Two Dimensional Real Forward FFT - Row Compact*

#### **C USAGE:**

```
float rv1[][2], cvn[2], cvo[][2];
long inc2, nr, nc;
rft2fr_(rv1, cvn, &inc2, cvo, &nr, &nc);
-or-
rft2fr(rv1, cvn, inc2, cvo, nr, nc);
```

#### **FORTRAN USAGE:**

```
real rv1()
complex cvn(), cvo()
integer*4 inc2, nr, nc
call rft2fr(rv1, cvn, inc2, cvo, nr, nc)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
cvn	(out) - Complex Nyquist vector
inc2	(inp) - Nyquist vector increment
cvo	(out) - Complex output vector
nr	(inp) - Number of rows in rv1
nc	(inp) - Number of columns in rv1

#### **SUBROUTINE DESCRIPTION:**

The two dimensional forward real FFT of the input vector is computed and stored in the output vector. The rows are assumed to be compact.

The two dimensional real FFT is defined as a series of forward real-to-complex FFTs down the columns, followed by forward complex-to-complex FFTs across the rows.

The real FFTs each produce a single complex Nyquist frequency output, which is stored in cvn. A final complex-to-complex FFT of cvn is computed.

#### **MATHEMATICAL EQUIVALENT:**

```
for i = 0 to nr-1:
    rfftf(rv1, 1, cvo, 2, cvn[i*inc2], nc)
for i = 0 to nc/2-1:
    cfftf(cvo, nc, cvo, nc, nr)

    cfftf(cvn, inc2, cvn, inc2, nr)
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Input data is assumed to be in row order (C)
- 2) Size is restricted to 2048 x 1024 (nr x nc)

## **rft2ic**

### *Two Dimensional Real Inverse FFT - Column Compact*

#### **C USAGE:**

```
float cv1[][2], cvn[2], rvo[][2];
long inc2, nr, nc;
rft2ic_(cv1, cvn, &inc2, rvo, &nr, &nc);
-or-
rft2ic(cv1, cvn, inc2, rvo, nr, nc);
```

#### **FORTRAN USAGE:**

```
complex cv1(), cvn()
real rvo()
integer*4 inc2, nr, nc
call rft2ic(cv1, cvn, inc2, rvo, nr, nc)
```

#### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
cvn	(inp) - Complex Nyquist vector
inc2	(inp) - Nyquist vector increment
rvo	(out) - Real output vector
nr	(inp) - Number of rows in rv1
nc	(inp) - Number of columns in rv1

#### **SUBROUTINE DESCRIPTION:**

The two dimensional inverse real FFT of the input vector is computed and stored in the output vector. The columns are assumed to be compact.

The two dimensional real FFT is defined as a series of inverse complex-to-complex FFTs across the rows and input vector cvn, followed by inverse complex-to-real FFTs down the columns.

The real inverse FFTs each require a single real Nyquist frequency input, which is read from the inverse FFT of cvn.

#### **MATHEMATICAL EQUIVALENT:**

```
for i = 0 to nr/2-1:
    cffti(cv1, nr, rvo, nr, nc);
    cffti(cvn, inc2, ctmp, 2, nc);
for i = 0 to nc-1:
    rffti(rvo, 2, ctmp[i], rvo, 1, nr);
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Input data is assumed to be in column order (FORTRAN)
- 2) Size is restricted to 2048 x 1024 (nr x nc).

## **rft2ir**

### *Two Dimensional Real Inverse FFT - Row Compact*

#### **C USAGE:**

```
float cv1[][2], cvn[2], rvo[][2];
long inc2, nr, nc;
rft2ir_(cv1, cvn, &inc2, rvo, &nr, &nc);
-or-
rft2ir(cv1, cvn, inc2, rvo, nr, nc);
```

#### **FORTRAN USAGE:**

```
complex cv1(), cvn()
real rvo()
integer*4 inc2, nr, nc
call rft2ir(cv1, cvn, inc2, rvo, nr, nc)
```

#### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
cvn	(inp) - Complex Nyquist vector
inc2	(inp) - Nyquist vector increment
rvo	(out) - Real output vector
nr	(inp) - Number of rows in rv1
nc	(inp) - Number of columns in rv1

#### **SUBROUTINE DESCRIPTION:**

The two dimensional inverse real FFT of the input vector is computed and stored in the output vector. The rows are assumed to be compact.

The two dimensional real FFT is defined as a series of inverse complex-to-complex FFTs across the rows and input vector cvn, followed by inverse complex-to-real FFTs down the columns.

The real inverse FFTs each require a single real Nyquist frequency input, which is read from the inverse FFT of cvn.

#### **MATHEMATICAL EQUIVALENT:**

```
for i = 0 to nc/2-1:
    cffti(cv1, nc, rvo, nc, nr);
    cffti(cvn, inc2, ctmp, 2, nc);
for i = 0 to nr-1:
    rffti(rvo, 2, ctmp[i], rvo, 1, nc);
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Input data is assumed to be in row order (C)
- 2) Size is restricted to 2048 x 1024 (nr x nc).

## **rges**

*Find Location of First Element Greater/Equal to a Scalar*

### **C USAGE:**

```
float rv1[], rs1;  
long incl, idx, n;  
rges_(rv1, &incl, &rs1, &n, &idx);  
-or-  
rges(rv1, incl, &rs1, n, &idx);
```

### **FORTRAN USAGE:**

```
real rv1(), rs1  
integer*4 incl, idx, n  
call rges(rv1, incl, rs1, n, idx)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rs1	(inp) - Real scalar input
n	(inp) - Number of points in output vector
idx	(out) - Vector index

### **SUBROUTINE DESCRIPTION:**

The vector rv1 is searched from the beginning for an element greater or equal to the scalar rs1. The position of the found element is returned in idx, but is not a traditional offset. It is the position within the examined elements only and is dependent on the increment used. The first position is 0. If no element is found meeting the criteria, -1 is returned for idx.

### **MATHEMATICAL EQUIVALENT:**

```
i=0  
idx=0  
while i < n:  
    if rv1[i] >= rs1 then break  
    i = i + incl  
    idx = idx + 1
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **rgts**

*Find Location of First Element Greater Than a Scalar*

### **C USAGE:**

```
float rv1[], rs1;  
long incl, idx, n;  
rgts_(rv1, &incl, &rs1, &n, &idx);  
-or-  
rgts(rv1, incl, &rs1, n, &idx);
```

### **FORTRAN USAGE:**

```
real rv1(), rs1  
integer*4 incl, idx, n  
call rgts(rv1, incl, rs1, n, idx)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rs1	(inp) - Real scalar input
n	(inp) - Number of points in output vector
idx	(out) - Vector index

### **SUBROUTINE DESCRIPTION:**

The vector rv1 is searched from the beginning for an element greater than the scalar rs1. The position of the found element is returned in idx, but is not a traditional offset. It is the position within the examined elements only and is dependent on the increment used. The first position is 0. If no element is found meeting the criteria, -1 is returned for idx.

### **MATHEMATICAL EQUIVALENT:**

```
i=0  
idx=0  
while i < n:  
    if rv1[i] > rs1 then break  
    i = i + incl  
    idx = idx + 1
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **rlts**

*Find Location of First Element Less Than a Scalar*

### **C USAGE:**

```
float rv1[], rs1;  
long incl, idx, n;  
rlts_(rv1, &incl, &rs1, &n, &idx);  
-or-  
rlto(rv1, incl, &rs1, n, &idx);
```

### **FORTRAN USAGE:**

```
real rv1(), rs1  
integer*4 incl, idx, n  
call rlto(rv1, incl, rs1, n, idx)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rs1	(inp) - Real scalar input
n	(inp) - Number of points in output vector
idx	(out) - Vector index

### **SUBROUTINE DESCRIPTION:**

The vector rv1 is searched from the beginning for an element less than the scalar rs1. The position of the found element is returned in idx, but is not a traditional offset. It is the position within the examined elements only and is dependent on the increment used. The first position is 0. If no element is found meeting the criteria, -1 is returned for idx.

### **MATHEMATICAL EQUIVALENT:**

```
i = 0  
idx = 0  
while i < n:  
    if rv1[i] < rs1 then break  
    i = i + incl  
    idx = idx + 1
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **rmax**

*Find the Maximum Value and its Location*

### **C USAGE:**

```
float rv1[], rso;
long incl, idx, n;
rmax_(rv1, &incl, &rso, &idx, &n);
-or-
rmax(rv1, incl, &rso, &idx, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso
integer*4 incl, idx, n
call rmax(rv1, incl, rso, idx, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rso	(out) - Maximum value found
idx	(out) - Location of maximum
n	(inp) - Number of points to scan

### **SUBROUTINE DESCRIPTION:**

The vector rv1 is searched and the maximum value found is returned in rso. The position of this element is returned in idx, but is not a traditional offset. It is the position within the examined elements only and is dependent on the increment used. The first position is 0.

### **MATHEMATICAL EQUIVALENT:**

```
i = 0
idx = 0
rso = rv1[0]
while i < n:
    if rv1[i] > rso then rso = rv1[i]
    i = i + incl
    idx = idx + i
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **rmaxmg**

*Find the Maximum Magnitude*

### **C USAGE:**

```
float rv1[], rso;
long incl, idx, n;
rmaxmg_(rv1, &incl, &rso, &idx, &n);
-or-
rmaxmg(rv1, incl, &rso, &idx, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso
integer*4 incl, idx, n
call rmaxmg(rv1, incl, rso, idx, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rso	(out) - Maximum magnitude found
idx	(out) - Index of element with max magnitude
n	(inp) - Number of points in output vector

### **SUBROUTINE DESCRIPTION:**

The vector rv1 is searched from the beginning for the element with the greatest magnitude. The position of the found element is returned in idx, but is not a traditional offset. It is the position within the examined elements only and is dependent on the increment used. The first position is 0.

### **MATHEMATICAL EQUIVALENT:**

```
i = 0
idx = 0
rso = |rv1[0]|
while i < n:
    if |rv1[i]| > rso then rso = |rv1[i]|
    i = i + incl
    idx = idx + 1
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **rmin**

*Find the Minimum Value and its Position*

### **C USAGE:**

```
float rv1[], rso;  
long incl, idx, n;  
rmin_(rv1, &incl, &rso, &idx, &n);  
-or-  
rmin(rv1, incl, &rso, &idx, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso  
integer*4 incl, idx, n  
call rmin(rv1, incl, rso, idx, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rso	(out) - Minimum value found
idx	(out) - Position of minimum
n	(inp) - Number of points to scan

### **SUBROUTINE DESCRIPTION:**

The vector rv1 is searched from the beginning for the element with the least value. The position of the found element is returned in idx, but is not a traditional offset. It is the position within the examined elements only and is dependent on the increment used. The first position is 0.

### **MATHEMATICAL EQUIVALENT:**

```
i = 0  
idx = 0  
rso = rv1[0]  
while i < n:  
    if rv1[i] < rso then rso = rv1[i]  
    i = i + incl  
    idx = idx + 1
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **rminmg**

*Find the Minimum Magnitude*

### **C USAGE:**

```
float rv1[], rso;  
long incl, idx, n;  
rminmg_(rv1, &incl, &rso, &idx, &n);  
-or  
rminmg(rv1, incl, &rso, &idx, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso  
integer*4 incl, idx, n  
call rminmg(rv1, incl, roo, idx, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rso	(out) - Minimum magnitude found
idx	(out) - Position of element found
n	(inp) - Number of points in output vector

### **SUBROUTINE DESCRIPTION:**

The vector rv1 is searched from the beginning for the element with the least magnitude. The position of this element is returned in idx, but is not a traditional offset. It is the position within the examined elements only and is dependent on the increment used. The first position is 0.

### **MATHEMATICAL EQUIVALENT:**

```
i = 0  
idx = 0  
rso = |rv1[0]|  
while i < n:  
    if |rv1[i]| < rso then rso = |rv1[i]|  
    i = i + incl  
    idx = idx + 1
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **rmsqv**

*Root Mean Square of a Real Vector*

### **C USAGE:**

```
float rv1[], rso;  
long incl, n;  
rmsgv_(rv1, &incl, &rso, &n);  
-or-  
rmsqv(rv1, incl, &rso, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso  
integer*4 incl, n  
call rmsqv(rv1, incl, rso, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rso	(out) - Real scalar output
n	(inp) - Number of points in output vector

### **SUBROUTINE DESCRIPTION:**

The square root of the mean value of the squares of the processed elements of rv1 is returned in rso.

### **MATHEMATICAL EQUIVALENT:**

$$rso = \text{SQRT} \{ \text{SUM} \{ rv1[i*incl] * rv1[i*incl] \} / n \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **rnes**

*Find Position of First Element Not Equal to a Scalar*

### **C USAGE:**

```
float rv1[], rs1;  
long incl, idx, n;  
rnes_(rv1, &incl, &rs1, &n, &idx);  
-or-  
rnes(rv1, incl, &rs1, n, &idx);
```

### **FORTRAN USAGE:**

```
real rv1(), rs1  
integer*4 incl, idx, n  
call rnes(rv1, incl, rs1, n, idx)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rs1	(inp) - Real scalar input
n	(inp) - Number of points to examine
idx	(out) - Vector index

### **SUBROUTINE DESCRIPTION:**

The vector rv1 is searched from the beginning for an element not equal to the scalar rs1. The position of the found element is returned in idx, but is not a traditional offset. It is the position within the examined elements only and is dependent on the increment used. The first position is 0. If no element is found meeting the criteria, -1 is returned for idx.

### **MATHEMATICAL EQUIVALENT:**

```
i = 0  
idx = 0  
while i < n:  
    if rv1[i] <> rs1 then break  
    i = i + incl  
    idx = idx + 1
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **rsve**

## *Running Sum of Real Vector*

### **C USAGE:**

```
float rv1[], rs1, rvo[];
long  incl, inco, no, nw;
rsve_(rv1, &incl, &rs1, rvo, &inco, &no, &nw);
-or-
rsve(rv1, incl, &rs1, rvo, inco, no, nw);
```

### **FORTRAN USAGE:**

```
real rv1(), rs1, rvo()
integer*4 incl, inco, no, nw
call rsve(rv1, incl, rs1, rvo, inco, no, nw)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rs1	(inp) - Real scalar input
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
no	(inp) - Number of sums in output vector
nw	(inp) - Number of points to sum in running window

### **SUBROUTINE DESCRIPTION:**

The running sum of the elements of rv1, scaled by rs1, is written to rvo. Each output element is the sum of nw elements centered around the corresponding input point.

The first nw/2 output points are computed over a variable window with a starting width of nw/2 elements, increasing by one for each output point, to a width of nw. The last nw/2 output points are similarly computed over a window tapering down from nw to nw/2.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rs1 * \text{SUM}\{rv1[(i+j)*incl]\} \text{ for } i=0 \text{ to } no-1, \text{ for } j = -nw/2 \text{ to } nw/2$$
  
(i+j) is restricted to the range 0 to no-1

### **RESTRICTIONS & SPECIAL CONDITIONS:**

rv1 must contain at least nw/2+1 points.  
nw is restricted to the size of cache. (i860 only)

## **sve**

## *Sum of Real Vector*

### **C USAGE:**

```
float rv1[], rso;  
long incl, n;  
sve_(rv1, &incl, &rso, &n);  
-or-  
sve(rv1, incl, &rso, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso  
integer*4 incl, n  
call sve(rv1, incl, rso, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rso	(out) - Real scalar output
n	(inp) - Number of points to sum

### **SUBROUTINE DESCRIPTION:**

The sum of the elements of rv1 is written to rso.

### **MATHEMATICAL EQUIVALENT:**

$$rso = \text{SUM} \{rv1[i*incl]\}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## shphu

## *Schafer's Phase Unwrapping*

### C USAGE:

```
float rv1[], rvo[];
float rs1;
long incl, inco, n;
shphu_(rv1, &incl, &rs1, rvo, &inco, &n);
-or-
shphu(rv1, incl, &rs1, rvo, inco, n);
```

### FORTRAN USAGE:

```
real rv1(), rvo(), rs1
integer*4 incl, inco, n
call shphu(rv1, incl, rs1, rvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rs1	(inp) - Real comparative scalar
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points in output vector

### SUBROUTINE DESCRIPTION:

Input vector rv1 is assumed to be a vector of phase angles, in units of radians. The Output vector contains the phase angles with multiples of  $2\pi$  added or subtracted, based on the difference between adjoining points relative to an input threshold, rs1. This has the effect of "unwrapping" the discontinuities caused by phases having being limited to magnitudes of  $2\pi$  during a previous computation.

### MATHEMATICAL EQUIVALENT:

```
rvo[i*inco] = rv1[i*incl] + cfac
where correction factor (cfac) is: cfac[1] = 0
if rv1[(i+1)*incl] - rv1[i*incl] > 2π - rs1, then cfac[i+1] = cfac[i] - 2π
if rv1[(i+1)*incl] - rv1[i*incl] < rs1 - 2π, then cfac[i+1] = cfac[i] + 2π
else cfac[i+1] = cfac[i]
```

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## shphuf

*Schafer's Phase Unwrapping, Fraction of a Circle Argument*

### C USAGE:

```
float rv1[], rvo[];
float rs1;
long incl, inco, n;
shphuf_(rv1, &incl, &rs1, rvo, &inco, &n);
-or-
shphuf(rv1, incl, &rs1, rvo, inco, n);
```

### FORTRAN USAGE:

```
real rv1(), rvo(), rs1
integer*4 incl, inco, n
call shphuf(rv1, incl, rs1, rvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rs1	(inp) - Real comparative scalar
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points in output vector

### SUBROUTINE DESCRIPTION:

Input vector rv1 is assumed to be a vector of phase angles, in units of fractions of a circle. The output vector contains the phase angles with multiples of 1.0 added or subtracted, based on the difference between adjoining points relative to an input threshold, rs1. This has the effect of "unwrapping" the discontinuities caused by phases having being limited to magnitudes of 1.0 by a previous computation.

### MATHEMATICAL EQUIVALENT:

```
rvo[i*inco] = rv1[i*incl] + cfac
where correction factor (cfac) is:
cfac[1] = 0
if rv1[(i+1)*incl] - rv1[i*incl] > 1.0-rs1, then cfac[i+1] = cfac[i]-1.0
if rv1[(i+1)*incl] - rv1[i*incl] < rs1-1.0, then cfac[i+1]=cfac[i] +1.0
else cfac[i+1] = cfac[i]
```

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## **sn2**

*Sum the Squared Difference Between Two Vectors*

### **C USAGE:**

```
float rv1[], rv2[], rso;
long incl, inc2, n;
sn2_(rv1, &incl, rv2, &inc2, &rso, &n);
-or-
sn2(rv1, incl, rv2, inc2, &rso, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rso
integer*4 incl, inc2, n
call sn2(rv1, incl, rv2, inc2, rso, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment for rv1
rv2	(inp) - Second real input vector 2
inc2	(inp) - Vector increment for rv2
rso	(out) - Real scalar output
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The sum of the squares of the differences between vectors rv1 and rv2 is written to rso.

### **MATHEMATICAL EQUIVALENT:**

$$rso = \text{SUM} \{ (rv1[i*inc1] - rv2[i*inc2])^2 \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **svdiv**

*Divide Scalar by Vector*

### **C USAGE:**

```
float rv1[], rs1, rvo[];
long incl, inco, n;
svdiv_(&rs1, rv1, &incl, rvo, &inco, &n);
-or-
svdiv(&rs1, rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rs1, rvo()
integer*4 incl, inco, n
call svdiv(rs1, rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rs1	(inp) - Real scalar dividend
rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Divides the real scalar by the selected points in the input vector and stores the result in the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rs1 / rv1[i*incl], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

No explicit check is made for division by zero.

## **svemg**

## *Sum of Vector Magnitudes*

### **C USAGE:**

```
float rv1[], rso;  
long incl, n;  
svemg_(rv1, &incl, &rso, &n);  
-or-  
svemg(rv1, incl, &rso, n);
```

### **FORTRAN USAGE:**

```
real rv1(),*rso  
integer*4 incl, n  
call svemg(rv1, incl, rso, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rso	(out) - Real scalar output
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Sum of absolute values of rv1 is returned in rso.

### **MATHEMATICAL EQUIVALENT:**

$$rso = \text{SUM} \{ |rv1[i*incl]| \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **svsq**

*Sum of Vector Elements Squared*

### **C USAGE:**

```
float rv1[], rso;  
long incl, n;  
svsq_(rv1, &incl, &rso, &n);  
-or-  
svsq(rv1, incl, &rso, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso  
integer*4 incl, n  
call svsq(rv1, incl, rso, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rso	(out) - Real scalar output
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Sum of the squares of elements of rv1 is returned in rso.

### **MATHEMATICAL EQUIVALENT:**

$$rso = \text{SUM} \{ rv1[i*incl]^2 \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **svessq**

*Sum of Vector Element Signed Squares*

### **C USAGE:**

```
float rv1[], rso;
long incl, n;
svessq_(rv1, &incl, &rso, &n);
-or-
svessq(rv1, incl, &rso, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rso
integer*4 incl, n
call svessq(rv1, incl, rso, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rso	(out) - Real scalar output
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The sum of the signed squares of the elements of rv1 is written to rso.

### **MATHEMATICAL EQUIVALENT:**

$$rso = \text{SUM} \{ rv1[i*incl] * |rv1 (i*incl)| \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **tconv**

## *Tapered Convolution*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long inc1, inc2, inco, no, n2, n1;
tconv_(rv1, &inc1, rv2, &inc2, rvo, &inco, &no, &n2, &n1);
-or-
tconv(rv1, inc1, rv2, inc2, rvo, inco, no, n2, n1);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 inc1, inc2, no, n2, n1
call tconv(rv1, inc1, rv2, inc2, rvo, inco, no, n2, n1)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Second real input vector 2
inc2	(inp) - Vector increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
no	(inp) - Number of points to output
n2	(inp) - Number of points in rv2 (kernel)
n1	(inp) - Number of points in rv1

### **SUBROUTINE DESCRIPTION:**

tconv convolves the vector rv1 with the vector rv2 or correlates vectors rv1 with rv2 depending on the sign of inc1 and inc2. Vector rv1 is treated as padded with zeroes as necessary to allow the tapering effect to occur.

### **MATHEMATICAL EQUIVALENT:**

for  $k = 0$  to  $no-1$   
 $rvo(k*inco) = \text{SUM} \{ rv1[inc1*(m+k)]*rv2[inc2] \}$ , for  $m = 0$  to  $L$   
where  $L = \text{minimum of } (n2-1) \text{ or } (n2-k)$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **trans**

*Complex Vector Divided by Real Vector (Transfer)*

### **C USAGE:**

```
float rv1[], cv1[][2], cvo[][2];
long n;
trans_(rv1, cv1, cvo, &n);
-or-
trans(rv1, cv1, cvo, n);
```

### **FORTRAN USAGE:**

```
real*4 rv1()
complex cv1(), cvo()
integer*4 n
call trans(rv1, cv1, cvo, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
cv1	(inp) - Complex input vector
cvo	(out) - Complex output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Each element of the complex input vector, cv1, is divided by the corresponding element in the real vector, rv1, and copied to the complex output vector, cvo. (This routine is included only for backwards compatibility with other subroutine libraries. For new programs, use crvdiv.)

### **MATHEMATICAL EQUIVALENT:**

$$cvo[i] = cv1[i]/rv1[i] \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

All vectors must be compact.

## **vaam**

*Vector Add, Add, and Multiply*

### **C USAGE:**

```
float rv1[], rv2[], rv3[], rv4[], rvo[];
long inc1, inc2, inc3, inc4, inco, n;
vaam_(rv1, &inc1, rv2, &inc2, rv3, &inc3, rv4, &inc4, rvo, &inco, &n);
-or-
vaam(rv1, inc1, rv2, inc2, rv3, inc3, rv4, inc4, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rv3(), rv4(), rvo()
integer*4 inc1, inc2, inc3, inc4, inco, n
call vaam(rv1, inc1, rv2, inc2, rv3, inc3, rv4, inc4, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rv3	(inp) - Real input vector 3
inc3	(inp) - Vector increment for rv3
rv4	(inp) - Real input vector 4
inc4	(inp) - Vector increment for rv4
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of output samples to process

### **SUBROUTINE DESCRIPTION:**

Adds real input vectors rv1 and rv2, then multiplies by the sum of input vectors rv3 and rv4, and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$\text{rvo}[i*\text{inco}] = (\text{rv1}[i*\text{inc1}] + \text{rv2}[i*\text{inc2}]) \\ * (\text{rv3}[i*\text{inc3}] + \text{rv4}[i*\text{inc4}]), \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vabs**

## *Vector Absolute Value*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vabs_(rv1, &incl, rvo, &inco, n);
-or-
vabs(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vabs(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the absolute value of each element in the real input vector and places the results in the real output vector.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco]=|rv1[i*incl]|, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vacos**

## *Vector Arccosine*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vacos_(rv1, &incl, rvo, &inco, &n);
-or-
vacos(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vacos(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the arccosine of each element in the real input vector rv1 and places the result (in radians from 0 to  $\pi$ ) in the real output vector, rvo.

### **MATHEMATICAL EQUIVALENT:**

$rvo[i*inco]=ARCCOS(rv1[i*incl])$ , for  $i=0$  to  $n-1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vadd**

## *Add Two Vectors*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long inc1, inc2, inco, n;
vadd_(rv1, &inc1, rv2, &inc2, rvo, &inco, &n);
-or-
vadd(rv1, inc1, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 inc1, inc2, inco, n
call vadd(rv1, inc1, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the first and second input vectors are added and placed into the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*inc1] + rv2[i*inc2], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vaint**

*Vector Align to Integer*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vaint_(rv1, &incl, rvo, &inco, &n);
-or-
vaint(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vaint(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Truncates each element of the real input vector rv1 and places the integer portion in the real output vector rvo.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{TRUNC}\{ rv1[i*incl] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **vam**

## *Vector Add and Multiply*

### **C USAGE:**

```
float rv1[], rv2[], rv3[], rvo[];
long inc1, inc2, inc3, inco, n;
vam_(rv1, &inc1, rv2, &inc2, rv3, &inc3, rvo, &inco, &n);
-or-
vam(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rv3(), rvo()
integer*4 inc1, inc2, inc3, inco, n
call vam(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rv3	(inp) - Real input vector 3
inc3	(inp) - Vector increment for rv3
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Adds real input vectors rv1 and rv2, then multiplies by input vector rv3, and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = (rv1[i*inc1] + rv2[i*inc2]) * rv3[i*inc3], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vanint**

*Vector Align to Nearest Integer*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vanint_(rv1, &incl, rvo, &inco, &n);
-or-
vanint(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vanint(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Input vector increment
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Rounds each element of the real input vector rv1 and places the integer portion in the real output vector rvo. If the fractional portion is greater than or equal to .5, the output is rounded to the next higher integer.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{ROUND}\{ rv1[i*incl] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## vasbm

## *Vector Add, Subtract, and Multiply*

### C USAGE:

```
float rv1[], rv2[], rv3[], rv4[], rvo[];
long inc1, inc2, inc3, inc4, inco, n;
vasbm_(rv1, &inc1, rv2, &inc2, rv3, &inc3, rv4, &inc4, rvo, &inco, &n);
-or-
vasbm(rv1, inc1, rv2, inc2, rv3, inc3, rv4, inc4, rvo, inco, n);
```

### FORTRAN USAGE:

```
real rv1(), rv2(), rv3(), rv4(), rvo()
integer*4 inc1, inc2, inc3, inc4, inco, n
call vasbm(rv1, inc1, rv2, inc2, rv3, inc3, rv4, inc4, rvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rv3	(inp) - Real input vector 3
inc3	(inp) - Vector increment for rv3
rv4	(inp) - Real input vector 4
inc4	(inp) - Vector increment for rv4
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### SUBROUTINE DESCRIPTION:

Adds real input vectors rv1 and rv2, then multiplies by the difference of input vectors rv3 and rv4, and puts the result in output real vector rvo for n points.

### MATHEMATICAL EQUIVALENT:

$$\text{rvo}[i*\text{inco}] = (\text{rv1}[i*\text{inc1}] + \text{rv2}[i*\text{inc2}]) \\ *(\text{rv3}[i*\text{inc3}] - \text{rv4}[i*\text{inc4}]), \text{ for } i=0 \text{ to } n-1$$

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## vasin

## Vector Arcsine

### C USAGE:

```
float rv1[], rvo[];
long incl, inco, n;
vasin_(rv1, &incl, rvo, &inco, &n);
-or-
vasin(rv1, incl, rvo, inco, n);
```

### FORTRAN USAGE:

```
real rv1(), rvo()
integer*4 incl, inco, n
call vasin(rv1, incl, rvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### SUBROUTINE DESCRIPTION:

Computes the arcsine of the each element of the real input vector and places the result in the real output vector. Results are in radians, from  $-\pi/2$  to  $+\pi/2$ .

### MATHEMATICAL EQUIVALENT:

$$rvo[i*inco] = \text{ARCSIN}\{rv1[i*incl]\}, \text{ for } i=0 \text{ to } n-1$$

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## **vasm**

## *Vector Add and Scalar Multiply*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, n;
float rs1;
vasm_(rv1, &incl, rv2, &inc2, &rs1, rvo, &inco, &n);
-or-
vasm(rv1, incl, rv2, inc2, rs1, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, inco, n
real rs1
call vasm(rv1, incl, rv2, inc2, rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rs1	(inp) - Real scalar multiplier
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Adds the input real vector rv1 to real scalar rs1, then multiplies by real scalar rs1, and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = (rv1[i*incl] + rv2[i*inc2]) * rs1, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vatan**

## *Vector Arctangent*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vatan_(rv1, &incl, rvo, &inco, &n);
-or-
vatan(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vatan(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the arctangent of each element of the real input vector and places the result in the real output vector. Results are in radians, from  $-\pi/2$  to  $+\pi/2$ .

### **MATHEMATICAL EQUIVALENT:**

$rvo[i*inco] = \text{ARCTAN}\{rv1[i*incl]\}$ , for  $i=0$  to  $n-1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vatan2**

### *Vector Arctangent of Two Arguments*

#### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long inc1, inc2, inco, n;
vatan2_(rv1, &inc1, rv2, &inc2, rvo, &inco, &n);
-or-
vatan2(rv1, inc1, rv2, inc2, rvo, inco, n);
```

#### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 inc1, inc2, inco, n
call vatan2(rv1, inc1, rv2, inc2, rvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

Computes the arctangent of the fraction created by dividing each element of the first real input vector by the corresponding element of the second real input vector. The result is placed in the real output vector. Results are in radians, from  $-\pi/2$  to  $+\pi/2$ .

#### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{ARCTAN} \{rv1[i*inc1] / rv2[i*inc2]\}, \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

The output vector will be zero for elements where either input vector is zero, i.e. division by zero results in zero rather than an error.

## **vatan2f**

### *Vector Arctangent of Two Arguments in Fractions*

#### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, n;
vatan2f_(rv1, &incl, rv2, &inc2, rvo, &inco, &n);
-or-
vatan2f(rv1, incl, rv2, inc2, rvo, inco, n);
```

#### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, inco, n
call vatan2f(rv1, incl, rv2, inc2, rvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

Computes the arctangent of the fraction created by dividing each element of the first real input vector by the corresponding element of the second real input vector. The result is placed in the real output vector. Results are in fractions of a circle, from -0.5 to +0.5.

#### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{ARCTAN} \{rv1[i*incl] / rv2[i*inc2]\} / 2\pi, \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

The output vector will be zero for elements where either input vector is zero, i.e. division by zero results in zero rather than an error.



## **vavexp**

## *Vector Exponential Averaging*

### **C USAGE:**

```
float rv1[], rvo[l, rs1;  
long *incl, *inco, *n;  
vavexp_(rv1, &incl, &rs1, rvo, &inco, &n);  
-or-  
vavexp(rv1, incl, &rs1, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo(), rs1  
integer*4 incl, inco, n  
call vavexp(rv1, incl, rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment for rv1
rs1	(inp) - Real input scalar
rvo	(i/o) - Real output vector
inco	(out) - Output vector increment
n	(inp) - Number of points in output vector

### **SUBROUTINE DESCRIPTION:**

Calculates the exponential average of vector rvo and vector rv1 using rs1 as a time constant.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rvo[i*inco]*(rs1-1)/rs1 + rv1[i*incl]*1/rs1, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vavlin**

## *Vector Linear Averaging*

### **C USAGE:**

```
float rv1[], rvo[], rs1;
long *incl, *inco, *n;
vavlin_(rv1, &incl, &rs1, rvo, &inco, &n);
-or-
vavlin(rv1, incl, &rs1, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo(), rs1
integer*4 incl, inco, n
call vavlin(rv1, incl, rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment for rv1
rs1	(inp) - Real input scalar
rvo	(i/o) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of output points

### **SUBROUTINE DESCRIPTION:**

Calculates the linear average of vector rvo and vector rv1 using rs1 as the number of vectors averaged over. The result is placed in rvo.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rvo[i*inco] * rs1 / (rs1 + 1) + rv1[i*incl] * 1 / (rs1 + 1), \text{ for } i = 0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vclip**

## *Vector Clip*

### **C USAGE:**

```
float rv1[], rvo[], rs1, rs2;  
long incl, inco, n;  
vclip_(rv1, &incl, &rs1, &rs2, rvo, &inco, &n);  
-or  
vclip(rv1, incl, &rs1, &rs2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo(), rs1, rs2  
integer*4 incl, inco, n  
call vclip(rv1, incl, rs1, rs2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rs1	(inp) - Real scalar minimum value
rs2	(inp) - Real scalar maximum value
rvo	(out) - Real vector output
inco	(inp) - Vector increment
n	(inp) - Number of points to process

### **SUBROUTINE DESCRIPTION:**

Compare vector points against minimum value rs1 and maximum value rs2, if less than rs1, use rs1, if greater than rs2, use rs2; else set rvo to input vector point value. Do this for n points.

### **MATHEMATICAL EQUIVALENT:**

$$\text{tmp} = \text{MAX} \{ \text{rv1}[\text{i} * \text{incl}], \text{rs1} \}, \text{ for } \text{i}=0 \text{ to } \text{n}-1$$
$$\text{rvo}[\text{i} * \text{inco}] = \text{MIN} \{ \text{tmp}, \text{rs2} \}$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vclr**

*Zero a Vector*

### **C USAGE:**

```
float rvo[];  
long inco, n;  
vclr_(rvo, &inco, &n);  
-or-  
vclr(rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rvo()  
integer*4 inco, n  
call vclr(rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rvo	(out) - Real output vector
inco	(inp) - Real output vector sample increment
n	(inp) - Number of points to clear

### **SUBROUTINE DESCRIPTION:**

Zeroes are written to the vector rvo.

### **MATHEMATICAL EQUIVALENT:**

$rv[i*inco] = 0.0$ , for  $i=0$  to  $n-1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vcmprs**

## *Vector Compress*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long inc1, inc2, inco, n;
vcmprs_(rv1, &inc1, rv2, &inc2, rvor &inco, &n);
-or-
vcmprs(rv1, inc1, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 inc1, inc2, inco, n
call vcmprs(rv1, inc1, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector Increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector Increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Each point in input vector rv2 is compared to zero. If the value in rv2 is non-zero, the corresponding point from input vector rv1 is put into corresponding point in output vector rvo.

### **MATHEMATICAL EQUIVALENT:**

```
j = 0
for i = 0 to n-1:
    if rv2[j*inc2] <> 0 then
        rvo[j*inco] = rv1[i*inc1]
    j = j + 1
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **VCOS**

## *Vector Cosine*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vcos_(rv1, &incl, rvo, &inco, &n);
-or-
vcos(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vcos(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the cosine of each element in the real input vector and places the result in the real output vector. The elements of the input vector are assumed to be in radians.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco]=\cos\{rv1[i*incl]\}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vcosf**

### *Vector Cosine in Fractions of a Circle*

#### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vcosf_(rv1, &incl, rvo, &inco, &n);
-or
vcosf(rv1, incl, rvo, inco, n);
```

#### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n call
vcosf(rv1, incl, rvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

Computes the cosine of each element in the real input vector and places the result in the real output vector. The elements of the input vector are assumed to be in fractions of a circle.

#### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \cos \{rv1[i*incl] * 2\pi\}, \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vdbpwr**

## *Vector Conversion to dB*

### **C USAGE:**

```
float rv1[], rvo[], *rs1;
long *incl, *inco, *n;
vdbpwr_(rv1, &incl, &rs1, rvo, &inco, &n);
-or-
vdbpwr(rv1, incl, &rs1, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo(), rs1
integer*4 incl, inco, n
call vdbpwr(rv1, incl, rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rs1	(inp) - Real scalar (0 dB value)
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the power in decibels of input real vector rv1, relative to real scalar rs1, and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = 10.0 * \log_{10}(rv1[i*incl]/rs1), \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

Input vector elements must be positive numbers.



## **vdiv**

*Divide One Vector by Another*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long inc1, inc2, inco, n;
vdiv(rv1, &inc1, rv2, &inc2, rvo, &inco, &n);
-or-
vdiv(rv1, inc1, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 inc1, inc2, inco, n
call vdiv(rv1, inc1, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the second input vector are divided by points from the first vector and the results are placed into the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv2[i*inc2] / rv1[i*inc1], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vdpsp**

### *Vector Convert Double to Single Precision*

#### **C USAGE:**

```
double dvl[];  
float rvo[];  
long incl, inco, n;  
vdpsp_(dvl, &incl, rvo, &inco, &n);  
-or-  
vdpsp(dvl, incl, rvo, inco, n);
```

#### **FORTRAN USAGE:**

```
real*8 dvl()  
real rvo()  
integer*4 incl, inco, n  
call vdpsp(dvl, incl, rvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

dvl	(inp) - Double precision real input vector
incl	(inp) - Increment for dvl
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of points to move

#### **SUBROUTINE DESCRIPTION:**

Converts the input double precision floating point vector dvl to single precision floating point and puts the result in output vector rvo.

#### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco]=SNGL\{dvl[i*incl]\}, \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## veud2

## *Vector Euclidean Distance*

### C USAGE:

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, n;
veuc12_(rv1, &incl, rv2, &inc2, rvo, &inco, &n);
-or-
veuc12(rv1, incl, rv2, inc2, rvo, inco, n);
```

### FORTRAN USAGE:

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, inco, n
call veuc12(rv1, incl, rv2, inc2, rvo, inco, n)
```

### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector 1
incl	(inp) - Vector Increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector Increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### SUBROUTINE DESCRIPTION:

Computes the square root of the sum of the squares of the two input vectors, rv1 and rv2 and stores the results in output vector rvo.

### MATHEMATICAL EQUIVALENT:

$$rvo[i*inco] = \text{SQRT}\{rv1[i*incl]^2 + rv2[i*inc2]^2\}, \text{ for } i=0 \text{ to } n-1.$$

### RESTRICTIONS & SPECIAL CONDITIONS:

None

## veud3

### *Vector Euclidean Distance (3 Dimensional)*

#### C USAGE:

```
float rv1[], rv2[], rv3[], rvo[];
long inc1, inc2, inc3, inco, n;
veucl3_(rv1, &inc1, rv2, &inc2, rv3, inc3, rvo, &inco, &n);
-or-
veucl3(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n);
```

#### FORTRAN USAGE:

```
real rv1(), rv2(), rv3(), rvo()
integer*4 inc1, inc2, inc3, inco, n
call veucl3(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n)
```

#### ARGUMENT DESCRIPTIONS:

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rv3	(inp) - Real input vector 3
inc3	(inp) - Vector increment for rv3
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

#### SUBROUTINE DESCRIPTION:

Computes the three dimensional distance of each point, using the three input vectors and writing the output to vector rvo.

#### MATHEMATICAL EQUIVALENT:

$$rvo[i*inco] = \text{SQRT}\{rv1[i*inc1]^2 + rv2[i*inc2]^2 + rv3[i*inc3]^2\}, \text{ for } i=0 \text{ to } n-1$$

#### RESTRICTIONS & SPECIAL CONDITIONS:

None

## **vexp**

## *Vector Exponentiation*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vexp_(rv1, &incl, rvo, &inco, &n);
-or-
vexp(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vexp(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment for rv1
rvo	(out) - Real output vector
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Raises the mathematical constant 'e' to the power of the elements in rv, for n points. The results are written to output vector rvo.

### **MATHEMATICAL EQUIVALENT:**

$rvo[i*inco] = e^{rv[i*inc]}$ , for  $i=0$  to  $n-1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vexp10**

## *Vector Base 10 Exponential*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vexp10_(rv1, &incl, rvo, &inco, &n);
-or-
vexp10(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vexp10(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the base 10 exponential of input real vector rv1 and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo(i*inco)=10^{rv1[i*incl]}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vexp2**

## *Vector Base 2 Exponential*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vexp2_(rv1, &incl, rvo, &inco, &n);
-or-
vexp2(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vexp2(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the base 2 exponential of input real vector rv1 and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = 2^{rv1[i*incl]}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vfill**

*Set a Vector in Memory to a Scalar Value*

### **C USAGE:**

```
float rvo[], rs1;  
long inco, n;  
vfill_(&rs1, rvo, &inco, &n);  
-or-  
vfill(&rs1, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rvo(), rs1  
integer*4 inco, n  
call vfill(rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rs1	(inp) - Real scalar
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of points to fill

### **SUBROUTINE DESCRIPTION:**

The scalar value rs1 is copied to the elements of output vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$rv[i*inco]=rs1$ , for  $i=0$  to  $n-1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **vfrac**

## *Vector Truncate to Fraction*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vfrac_(rv1, &incl, rvo, &inco, &n);
-or-
vfrac(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n call
vfrac(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Truncates each element of the real input vector and places the fractional portion in the real output vector. Result values will be between -1.0 and 1.0.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco]=rv1[i*incl]-TRUNC\{rv1[i*incl]\}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vfracn**

*Vector Truncate to Nearest Fraction*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vfracn_(rv1, &incl, rvo, &inco, &n);
-or-
vfracn(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vfracn(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Rounds each element of the real input vector and places the fractional portion in the real output vector. If the fractional portion is greater than or equal to .5, the output will reflect rounding to the next higher integer. The real output vector is in the range -0.5 to +0.5.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco]=rv1[i*incl]-ROUND\{rv1[i*incl]\}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vgathr**

## *Vector Gather*

### **C USAGE:**

```
float rv1[], rvo[];
long iv2[], inc2, inco, n;
vgathr_(rv1, iv2, &inc2, rvo, &inco, &n);
-or-
vgathr(rv1, iv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 iv2(), inc2, inco, n
call vgathr(rv1, iv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Input vector
iv2	(inp) - Vector containing indicies to rv1
inc2	(inp) - Increment for iv2
rvo	(out) - Output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the input vector iv2 are used as indices into input vector rv1 to select points to be placed into output vector rvo. The first point of vector rv1 has an index of 0.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco]=rv1[iv2[i*inc2]], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vgen**

*Generate a Vector in Memory*

### **C USAGE:**

```
float rvo[];  
float rs1, rs2;  
long inco, n;  
vgen_(&rs1, &rs2, rvo, &inco, &n);  
-or-  
vgen(&rs1, &rs2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rvo()  
real rs1, rs2  
integer*4 inco, n  
call vgen(rs1, rs2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rs1	(inp) - Starting value
rs2	(inp) - Ending value
rvo	(inp) - Real output vector
inco	(inp) - Output vector sample increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Generates a linear ramp from the value of rs1 to the value of rs2 and writes the results into rvo.

### **MATHEMATICAL EQUIVALENT:**

$$rv[i*inco] = rs1 + i*(rs2-rs1)/(n-1), \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **viadd**

## *Add Two Integer Vectors*

### **C USAGE:**

```
long iv1[], iv2[], ivo[];
long inc1, inc2, inco, n;
viadd_(iv1, &inc1, iv2, &inc2, ivo, &inco, &n);
-or-
viadd(iv1, inc1, iv2, inc2, ivo, inco, n);
```

### **FORTRAN USAGE:**

```
integer*4 iv1(), iv2(), ivo()
integer*4 inc1, inc2, inco, n
call viadd(iv1, inc1, iv2, inc2, ivo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

iv1	(inp) - First integer input vector
inc1	(inp) - Increment for first input vector
iv2	(inp) - Second integer input vector
inc2	(inp) - Increment for second input vector
ivo	(out) - Integer output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the first and second input vectors are added and placed into the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$\text{ivo}[i*\text{inco}] = \text{iv1}[i*\text{inc1}] + \text{iv2}[i*\text{inc2}], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **viand**

## *And Two Integer Vectors*

### **C USAGE:**

```
long iv1[], iv2[], ivo[];
long inc1, inc2, inco, n;
viand_(iv1, &inc1, iv2, &inc2, ivo, &inco, &n);
-or-
viand(iv1, inc1, iv2, inc2, ivo, inco, n);
```

### **FORTRAN USAGE:**

```
integer*4 iv1(), iv2(), ivo()
integer*4 inc1, inc2, inco, n
call viand(iv1, inc1, iv2, inc2, ivo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

iv1	(inp) - First integer input vector
inc1	(inp) - Increment for first input vector
iv2	(inp) - Second integer input vector
inc2	(inp) - Increment for second input vector
ivo	(out) - Integer output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the first and second input vectors are anded and placed into the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$ivo[i*inco] = iv1[i*inc1] \text{ AND } iv2[i*inc2], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **viars**

## *Integer Vector Arithmetic Right Shift*

### **C USAGE:**

```
long iv1[], ivo[];
long incl, ns, inco, n;
viars_(iv1, &incl, ns, ivo, &inco, &n);
-or-
viars(iv1, incl, ns, ivo, inco, n);
```

### **FORTRAN USAGE:**

```
integer*4 iv1(), iv2(), ivo()
integer*4 incl, ns, inco, n
call viars(iv1, incl, ns, ivo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

iv1	(inp) - First integer input vector
incl	(inp) - Increment for first input vector
ns	(inp) - Number of bits to shift
ivo	(out) - Integer output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the first vector are arithmetic right shifted and placed into the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$\text{ivo}[i*\text{inco}] = \text{ARITH\_RIGHT\_SHIFT} \{ \text{iv1}[i*\text{incl}], \text{ns} \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

Vector iv1 is not overwritten unless it is the same location as ivo, which is permissible as long as  $\text{inco} \leq \text{incl}$ .

## **viclip**

## *Vector Inverse Clip*

### **C USAGE:**

```
float rv1[], rvo[], rs1, rs2;  
long incl, inco, n;  
viclip_(rv1, &incl, &rs1, &rs2, rvo, &inco, &n);  
-or-  
viclip(rv1, incl, &rs1, &rs2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo(), rs1, rs2  
integer*4 incl, inco, n  
call viclip(rv1, incl, rs1, rs2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rs1	(inp) - Real scalar minimum value
rs2	(inp) - Real scalar maximum value
rvo	(out) - Real vector output
inco	(inp) - Vector increment
n	(inp) - Number of points in output vector

### **SUBROUTINE DESCRIPTION:**

Compare vector points; if negative, takes the smaller of input vector point rv1 and value rs1, if non-negative, takes the larger of input vector point rv1 and value rs2. The results are stored to output vector rvo.

### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:  
    if rv1[i*incl] < 0.0 then rvo[i*inco] = MIN{ rv1[i*incl], rs1}  
    if rv1[i*incl] >= 0.0 then rvo[i*inco] = MAX{ rv1[i*incl], rs2}
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **vils**

## *Integer Vector Left Shift*

### **C USAGE:**

```
long iv1[], ivo[];
long incl, is1, inco, n;
vils_(iv1, &incl, is1, ivo, &inco, &n);
-or-
vils(iv1, incl, is1, ivo, inco, n);
```

### **FORTRAN USAGE:**

```
integer*4 iv1(), iv2(), ivo()
integer*4 incl, is1, inco, n
call vils(iv1, incl, is1, ivo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

iv1	(inp) - First integer input vector
incl	(inp) - Increment for first input vector
is1	(inp) - Number of bits to shift
ivo	(out) - Integer output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the first vector are left shifted and placed into the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$\text{ivo}[i*\text{inco}] = \text{LEFT\_SHIFT} \{ \text{iv1}[i*\text{incl}], \text{ns} \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Vector iv1 is not overwritten unless it is the same location as ivo, which is permissible as long as inco <= to incl.
- 2)  $0 \leq \text{ns} \leq 32$

## **vimag**

*Extract Imaginary Part of Complex Vector*

### **C USAGE:**

```
float cv1[], rvo[];
long incl, inco, n;
vimag_(cv1, &incl, rvo, &inco, &n);
-or-
vimag(cv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1(), rvo()
integer*4 incl, inco, n
call vimag(cv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Input vector increment
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points in output vector

### **SUBROUTINE DESCRIPTION:**

Extract the imaginary part of complex vector cv1 and write results to output vector rvo.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{IMAG}\{ cv1[i*incl] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vimul**

## *Add Two Integer Vectors*

### **C USAGE:**

```
long iv1[], iv2[], ivo[];
long inc1, inc2, inco, n;
vimul_(iv1, &inc1, iv2, &inc2, ivo, &inco, &n);
-or-
vimul(iv1, inc1, iv2, inc2, ivo, inco, n);
```

### **FORTRAN USAGE:**

```
integer*4 iv1(), iv2(), ivo()
integer*4 inc1, inc2, inco, n
call vimul(iv1, inc1, iv2, inc2, ivo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

iv1	(inp) - First integer input vector
inc1	(inp) - Increment for first input vector
iv2	(inp) - Second integer input vector
inc2	(inp) - Increment for second input vector
ivo	(out) - Integer output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the first and second input vectors are multiplied and placed into the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$\text{ivo}[i*\text{inco}] = \text{iv1}[i*\text{inc1}] * \text{iv2}[i*\text{inc2}], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vindex**

*Vector Index, Truncate*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long inc2, inco, n;
vindex_(rv1, rv2, &inc2, rvo, &inco, &n);
-or-
vindex(rv1, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 inc1, inco, n
call vindex(rv1, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Input vector
rv2	(inp) - Vector containing indicies
inc2	(inp) - Increment for rv2
rvo	(out) - Output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the input vector rv2 are converted to integer by truncation. These integers are then used as indices into input vector rv1 for points to be placed into the output vector rvo.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[*inco] = rv1[INT\{rv2[*inc2]\}] \text{ , for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vineg**

## *Integer Vector Negate*

### **C USAGE:**

```
long iv1[], ivo[];
long incl, ns, inco, n;
vineg_(iv1, &incl, ns, ivo, &inco, &n);
-or-
vineg(iv1, incl, ns, ivo, inco, n);
```

### **FORTRAN USAGE:**

```
integer*4 iv1(), iv2(), ivo()
integer*4 incl, ns, inco, n
call vineg(iv1, incl, ns, ivo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

iv1	(inp) - First integer input vector
incl	(inp) - Increment for first input vector
ns	(inp) - Number of bits to shift
ivo	(out) - Integer output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the first vector are negated and placed into the output vector.

### **MATHEMATICAL EQUIVALENT:**

$ivo[i*inco] = -iv1[i*incl]$ , for  $i=0$  to  $n-1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

Vector iv1 is not overwritten unless it is the same location as ivo, which is permissible as long as  $inco \leq$  to  $incl$ .

## **vintb**

## *Vector Interpolate*

### **C USAGE:**

```
float rv1[], rv2[], rvo[], rs1;
long inc1, inc2, inco, n;
vintb_(rv1, &inc1, rv2, &inc2, &rs1, rvo, &inco, &n);
-or-
vintb(rv1, inc1, rv2, inc2, &rs1, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo(), rs1
integer*4 inc1, inc2, inco, n
call vintb(rv1, inc1, rv2, inc2, rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rs1	(inp) - Real scalar interpolation factor
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of output samples to process

### **SUBROUTINE DESCRIPTION:**

Does a linear interpolation between real input vectors rv1 and rv2 and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*inc1] + rs1*(rv2[i*inc2] - rv1[i*inc1]), \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vior**

*OR Two Integer Vectors*

### **C USAGE:**

```
long iv1[], iv2[], ivo[];
long inc1, inc2, inco, n;
viand_(iv1, &inc1, iv2, &inc2, ivo, &inco, &n);
-or-
viand(iv1, inc1, iv2, inc2, ivo, inco, n);
```

### **FORTRAN USAGE:**

```
integer*4 iv1(), iv2(), ivo()
integer*4 inc1, inc2, inco, n
call viand(iv1, inc1, iv2, inc2, ivo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

iv1	(inp) - First integer input vector
inc1	(inp) - Increment for first input vector
iv2	(inp) - Second integer input vector
inc2	(inp) - Increment for second input vector
ivo	(out) - Integer output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the first and second input vectors are logically OR'd and placed into the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$ivo[i*inco] = iv1[i*inc1] \text{ OR } iv2[i*inc2], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **virs**

## *Integer Vector Right Shift*

### **C USAGE:**

```
long iv1[], ivo[];
long incl, is1, inco, n;
virs_(iv1, &incl, is1, ivo, &inco, &n);
-or-
virs(iv1, incl, is1, ivo, inco, n);
```

### **FORTRAN USAGE:**

```
integer*4 iv1(), iv2(), ivo()
integer*4 incl, is1, inco, n
call virs(iv1, incl, is1, ivo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

iv1	(inp) - First integer input vector
incl	(inp) - Increment for first input vector
is1	(inp) - Number of bits to shift
ivo	(out) - Integer output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the first vector are right shifted and placed into the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$\text{ivo}[i*\text{inco}] = \text{RIGHT\_SHIFT} \{ \text{iv1}[i*\text{incl}], \text{is1} \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Vector iv1 is not overwritten unless it is the same location as ivo, which is permissible as long as  $\text{inco} \leq \text{incl}$ .
- 2)  $0 \leq \text{ns} \leq 32$



## **visub**

## *Subtract Two Integer Vectors*

### **C USAGE:**

```
long iv1[], iv2[], ivo[];
long inc1, inc2, inco, n;
visub_(iv1, &inc1, iv2, &inc2, ivo, &inco, &n);
-or-
visub(iv1, inc1, iv2, inc2, ivo, inco, n);
```

### **FORTRAN USAGE:**

```
integer*4 iv1(), iv2(), ivo()
integer*4 inc1, inc2, inco, n
call visub(iv1, inc1, iv2, inc2, ivo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

iv1	(inp) - First integer input vector
inc1	(inp) - Increment for first input vector
iv2	(inp) - Second integer input vector
inc2	(inp) - Increment for second input vector
ivo	(out) - Integer output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the first and second input vectors are subtracted and placed into the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$\text{ivo}[i*\text{inco}] = \text{iv1}[i*\text{inc1}] - \text{iv2}[i*\text{inc2}], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vixor**

### *Exclusive OR (XOR) Two Integer Vectors*

#### **C USAGE:**

```
long iv1[], iv2[], ivo[];
long incl, inc2, inco, n;
vixor_(iv1, &incl, iv2, &inc2, ivo, &inco, &n);
-or-
vixor(iv1, incl, iv2, inc2, ivo, inco, n);
```

#### **FORTRAN USAGE:**

```
integer*4 iv1(), iv2(), ivo()
integer*4 incl, inc2, inco, n
call vixor(iv1, incl, iv2, inc2, ivo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

iv1	(inp) - First integer input vector
incl	(inp) - Increment for first input vector
iv2	(inp) - Second integer input vector
inc2	(inp) - Increment for second input vector
ivo	(out) - Integer output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

The selected points from the first and second input vectors are exclusively ORed (XORed) and placed into the output vector.

#### **MATHEMATICAL EQUIVALENT:**

$$\text{ivo}[i*\text{inco}] = \text{iv1}[i*\text{incl}] \text{ XOR } \text{iv2}[i*\text{inc2}], \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vlim**

## *Vector Limit*

### **C USAGE:**

```
float rv1[], rvo[], rs1, rs2;  
long incl, inco, n;  
vlim_(rv1, &incl, &rs1, &rs2, rvo, &inco, &n);  
-or-  
vlim(rv1, incl, &rs1, &rs2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo(), rs1, rs2  
integer*4 incl, inco, n  
call vlim(rv1, incl, rs1, rs2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rs1	(inp) - Real scalar threshold
rs2	(inp) - Real magnitude of output
rvo,	(out) - Real vector output
inco	(out) - Vector increment
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Elements in output vector rvo are set to +rs2 or -rs2, depending on whether the corresponding elements in input vector rv1 are greater than or equal to input scalar rs1, or less than rs1, respectively.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rs2 * \text{SIGN}\{ rv1[i*incl] - rs1 \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vlint**

## *Vector Linear Interpolate*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long n1, inc2, inco, n;
vlint_(rv1, &n1, rv2, &inc2, rvo, &inco, &n);
-or-
vlint(rv1, n1, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 n1, inc2, inco, n
call vlint(rv1, n1, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real compact input vector
n1	(inp) - Integer size of vector rv1
rv2	(inp) - Second real input vector
inc2	(inp) - Increment for second input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

A linear interpolation is performed between selected points on input vector rv1 by using selected points of input vector rv2 as an index (whole number part) and interpolation factor (fractional part). Results are put into output vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    indx = INT{ rv2[i*inc2] }
    fac = rv2[i*inc2] - indx
    rvo=rv1[indx] + fac * (rv1[indx+1] - rv1[indx])
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

The truncated values in input vector rv2 must be less than the size of n1-2.

## **vlmerg**

## *Vector Logical Merge*

### **C USAGE:**

```
float rv1[], rv2[], rv3[], rvo[];
long inc1, inc2, inc3, inco, n;
vlmerg_(rv1, &inc1, rv2, &inc2, rv3, &inc3, rvo, &inco, n);
-or-
vlmerg(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rv3(), rvo()
integer*4 inc1, inc2, inc3, inco, n
call vlmerg(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment vector for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rv3	(inp) - Real input vector 3
inc3	(inp) - Vector increment for rv3
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Each element in output vector rvo is filled with either an element from input vector rv1, or from input vector rv2, based on the value of the elements in input vector rv3. The input vector selected will be rv1 if the corresponding element in rv3 is nonzero, or rv2 if the element in rv3 is zero.

### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    if rv3[i*inc3] <> 0.0 then rvo[i*inco] = rv1[i*inc1]
    else rvo[i*inco] = rv2[i*inc2]
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vlog**

## *Natural Logarithm of a Vector*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vlog_(rv1, &incl, rvo, &inco, &n);
-or-
vlog(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vlog(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment for rv1
rvo	(out) - Real output vector
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The natural logarithm of the points in vector rv1 are written to the output vector rvo.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{LOG}\{rv1[i*incl]\}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vlog10**

## *Vector Base 10 Logarithm*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vlog10_(rv1, &incl, rvo, &inco, &n);
-or-
vlog10(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vlog10(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the base 10 logarithm of input real vector rv1 and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco]=LOG_{10} \{ rv1[i*incl] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vlog2**

## *Vector Base 2 Logarithm*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vlog2_(rv1, &incl, rvo, &inco, &n);
-or-
vlog2(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vlog2(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the base 2 logarithm of input real vector rv1 and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco]=LOG\_2\{ rv1[i*incl] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **vma**

## *Vector Multiply and Add*

### **C USAGE:**

```
float rv1[], rv2[], rv3[], rvo[];
long inc1, inc2, inc3, inco, n;
vma_(rv1, &inc1, rv2, &inc2, rv3, &inc3, rvo, &inco, &n);
-or-
vma(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rv3(), rvo()
integer*4 inc1, inc2, inc3, inco, n
call vma(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rv3	(inp) - Real input vector 3
inc3	(inp) - Vector increment for rv3
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Multiplies real input vectors rv1 and rv2, adds input vector rv3, and puts the result in output vector rvo, for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*inc1] * rv2[i*inc2] + rv3[i*inc3], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vmax**

## *Vector Maximum*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, n;
vmax_(rv1, &incl, rv2, &inc2, rvo, &inco, &n);
-or-
vmax(rv1, incl, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, inco, n
call vmax(rv1, incl, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment 1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment 2
rvo	(out) - Real vector output
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Each element of output vector rvo is the larger of the corresponding element in input vector rv1 or rv2.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{MAX} \{ rv1[i*incl], rv2[i*inc2] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vmaxmg**

## *Vector Maximum Magnitude*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, n;
vmaxmg_(rv1, &incl, rv2, &inc2, rvo, &inco, &n);
-or-
vmaxmg(rv1, incl, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, inco, n
call vmaxmg(rv1, incl, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment 1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment 2
rvo	(out) - Real vector output
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Compare absolute value of two input vector points, take larger value and put into output vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{MAX} \{ |rv1[i*incl]|, |rv2[i*inc2]| \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vmin**

## *Vector Minimum*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long incl, inc2, inco, n;
vmin_(rv1, &incl, rv2, &inc2, rvo, &inco, &n);
-or-
vmin(rv1, incl, rv2, inc2, rvo, inco, W;
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 incl, inc2, inco, n
call vmin(rv1, incl, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment 1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment 2
rvo	(out) - Real vector output
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Compare two input vector points, take smaller value and put into output vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{MIN}\{rv1[i*incl], rv2[i*inc2]\}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vminmg**

## *Vector Minimum Magnitude*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long inc1, inc2, inco, n;
vminmg_(rv1, &inc1, rv2, &inc2, rvo, &inco, &n);
-or-
vminmg(rv1, inc1, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 inc1, inc2, inco, n
call vminmg(rv1, inc1, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment 1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment 2
rvo	(out) - Real vector output
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Compare absolute value of two input vector points, take smaller value and put it in output vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{MIN}\{ |rv1[i*inc1]|, |rv2[i*inc2]| \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vmma**

## *Vector Multiply, Multiply, and Add*

### **C USAGE:**

```
float rv1[], rv2[], rv3[], rv4[], rvo[];
long inc1, inc2, inc3, inc4, inco, n;
vmma_(rv1, &inc1, rv2, &inc2, rv3, &inc3, rv4, &inc4, rvo, &inco, &n);
-or-
vmma(rv1, inc1, rv2, inc2, rv3, inc3, rv4, inc4, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rv3(), rv4(), rvo()
integer*4 inc1, inc2, inc3, inc4, inco, n
call vmma(rv1, inc1, rv2, inc2, rv3, inc3, rv4, inc4, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment vector for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment vector for rv2
rv3	(inp) - Real input vector 3
inc3	(inp) - Vector increment vector for rv3
rv4	(inp) - Real input vector 4
inc4	(inp) - Vector increment vector for rv4
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Multiplies real input vectors rv1 and rv2, and adds it to the product of rv3 and rv4, and puts the result into output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = (rv1[i*inc1] * rv2[i*inc2]) + (rv3[i*inc3] * rv4[i*inc4]), \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vmmsb**

### *Vector Multiply, Multiply, and Subtract*

#### **C USAGE:**

```
float rv1[], rv2[], rv3[], rv4[], rvo[];
long inc1, inc2, inc3, inc4, inco, n;
vmmsb_(rv1, &inc1, rv2, &inc2, rv3, &inc3, rv4, &inc4, rvo, &inco, &n),
-or-
vmmsb(rv1, inc1, rv2, inc2, rv3, inc3, rv4, inc4, rvo, inco, n);
```

#### **FORTRAN USAGE:**

```
real rv1(), rv2(), rv3(), rv4(), rvo()
integer*4 inc1, inc2, inc3, inc4, inco, n
call vmmsb(rv1, inc1, rv2, inc2, rv3, inc3, rv4, inc4, rvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rv3	(inp) - Real input vector 3
inc3	(inp) - Vector increment for rv3
rv4	(inp) - Real input vector 4
inc4	(inp) - Vector increment for rv4
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

Multiplies real input vectors rv1 and rv2, and subtracts the product of rv3 and rv4. The results are placed into real output vector rvo.

#### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = (rv1[i*inc1] * rv2[i*inc2]) - (rv3[i*inc3] * rv4[i*inc4]), \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vmov**

*Copy One Vector to Another*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vmov_(rv1, &incl, rvo, &inco, &n);
-or-
vmov(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vmov(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment for rv1
rvo	(out) - Real output vector
inco	(inp) - Vector increment for rvo
n	(inp) - Number of points to copy

### **SUBROUTINE DESCRIPTION:**

The elements in vector rv1 are copied to vector rvo.

### **MATHEMATICAL EQUIVALENT:**

$rvo[i*inco] = rv1[i*incl]$ , for  $i=0$  to  $n-1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **vmsa**

## *Vector Multiply and Scalar Add*

### **C USAGE:**

```
float rv1[], rv2[], rvo[], rs1;
long inc1, inc2, inco, n;
vmsa_(rv1, &inc1, rv2, &inc2, &rs1, rvo, &inco, &n);
-or-
vmsa(rv1, inc1, rv2, inc2, &rs1, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo(), rs1
integer*4 inc1, inc2, inco, n
call vmsa(rv1, inc1, rv2, inc2, rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real vector one
inc1	(inp) - Increment for rv1
rv2	(inp) - Real vector two
inc2	(inp) - Increment for rv2
rs1	(inp) - Real scalar additive factor
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of points to move

### **SUBROUTINE DESCRIPTION:**

Multiplies real input vectors rv1 and rv2, adds real scalar rs1, and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*inc1] * rv2[i*inc2] + rs1, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vmsb**

## *Vector Multiply and Subtract*

### **C USAGE:**

```
float rv1[], rv2[], rv3[], rvo[];
long inc1, inc2, inc3, inco, n;
vmsb_(rv1, &inc1, rv2, &inc2, rv3, &inc3, rvo, &inco, &n);
-or-
vmsb(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rv3(), rvo()
integer*4 inc1, inc2, inc3, inco, n
call vmsb(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rv3	(inp) - Real input vector 3
inc3	(inp) - Vector increment for rv3
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Multiplies real input vectors rv1 and rv2, and subtracts input vector rv3, and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*inc1] * rv2[i*inc2] - rv3[i*inc3], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vmul**

## *Multiply Two Vectors*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long inc1, inc2, inco, n;
vmul_(rv1, &inc1, rv2, &inc2, rvo, &inco, &n);
-or-
vmul(rv1, inc1, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 inc1, inc2, inco, n
call vmul(rv1, inc1, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
inc1	(inp) - Increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The selected points from the first and second input vectors are multiplied and the products are placed into the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*inc1] * rv2[i*inc2]$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vnabs**

## *Vector Negative Absolute Value*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vnabs_(rv1, &incl, rvo, &inco, &n);
-or-
vnabs(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vnabs(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Takes the negation of the absolute value of each element in the real input vector and places it in the real output vector.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = -|rv1[i*incl]|$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vneg**

*Negate a Vector*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vneg_(rv1, &incl, rvo, &inco, &n);
-or-
vneg(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vneg(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real vector to negate
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Copies the negative of the desired points in the input vector to the output vector.

### **MATHEMATICAL EQUIVALENT:**

$rvo[i*inco] = -rv1[i*incl]$ , for  $i=0$  to  $n-1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vnmsa**

### *Vector Negative Multiply and Scalar Add*

#### **C USAGE:**

```
float rv1[], rv2[], rvo[], rs1;
long incl, inc2, inco, n;
vnmsa_(rv1, &incl, rv2, &inc2, &rs1, rvo, &inco, &n);
-or-
vrmsa(rv1, incl, rv2, inc2, &rs1, rvo, inco, n);
```

#### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo(), rs1
integer*4 incl, inc2, inco, n
call vnmsa(rv1, incl, rv2, inc2, rs1, rvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rs1	(inp) - Real scalar additive factor
rvo	(out) - Output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of points to move

#### **SUBROUTINE DESCRIPTION:**

Multiplies real input vectors rv1 and rv2, negates that and adds real scalar rs1, and puts the result in output real vector rvo for n points.

#### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = - (rv1[i*incl] * rv2[i*inc2]) + rs1, \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vpmerg**

## *Vector Positive Merge*

### **C USAGE:**

```
float rv1[], rv2[], rv3[], rvo[];
long inc1, inc2, inc3, inco, n;
vpmerg_(rv1, &inc1, rv2, &inc2, rv3, &inc3, rvo, &inco, &n);
-or-
vpmerg(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rv3(), rvo()
integer*4 inc1, inc2, inc3, inco, n
call vpmerg(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rv3	(inp) - Real input vector 3
inc3	(inp) - Vector increment for rv3
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Points from vectors rv1 and rv2 are transferred to vector rvo depending on the values in rv3. If an rv3 point is less than zero, the corresponding point in rv2 is transferred, otherwise the point is taken from rv1.

### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    if rv3[i*inc3] >=0.0 then rvo[i*inco] = rv1[i*inc1]
    else rvo[i*inco] = rv2[i*inc2]
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vpoly**

## *Vector Polynomial Evaluation*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long inc1, inc2, inco, n, n1;
vpoly_(rv1, &inc1, rv2, &inc2, rvo, &inco, &n, &n1);
-or-
vpoly(rv1, inc1, rv2, inc2, rvo, inco, n, n1);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 inc1, inc2, inco, n, n1
call vpoly(rv1, inc1, rv2, inc2, rvo, inco, n, n1)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Coefficient vector - highest order first
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Input vector 2
inc2	(inp) - Vector increment for rv2
rvo	(out) - Output vector
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process
n1	(inp) - Order of polynomial (size of rv1- 1)

### **SUBROUTINE DESCRIPTION:**

Does an element by element polynomial evaluation using elements of rv1 as the coefficients to be applied to elements of rv2. Results are stored to rvo. The coefficients are stored in rv1 in descending order, starting with rv1[0] as the highest order coefficient. The number of stored coefficients is 1 plus the order of the polynomial.

### **MATHEMATICAL EQUIVALENT:**

for i=0 to n-1:  
$$rvo[i*inco] = \sum_{j=0, n1} \{ rv1[j*inc1] * (rv2[i*inc2]^{(n1-j)}) \}$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

Do not specify the same vector for rv1 and rvo.



## **vqint**

## *Vector Quadratic Interpolate*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long n1, inc2, inco, n;
vqint_(rv1, &n1, rv2, &inc2, rvo, &inco, &n);
-or-
vqint(rv1, n1, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 n1, inc2, inco, n
call vqint(rv1, n1, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real compact input vector
n1	(inp) - Integer size of vector rv1
rv2	(inp) - Second real input vector
inc2	(inp) - Vector increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

A quadratic interpolation is performed between selected points on input vector rv1 by using selected points of input vector rv2 as an index (whole number part), interpolation factor (fractional part), and factor squared (fractional part times itself). Results are put into output vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

```
idx = INT{ rv2[i*inc2] }
fac = rv2[i*inc2] - idx
facsq = fac * fac
rvo = 0.5*(facsq-fac)*rv1[-1 +idx] + (1.0-facsq)*rv1[idx] + 0.5*(facsq+fac)*rv1[1+idx]
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) The truncated values in input vector rv2 cannot be greater than the number of elements in vector rv1 (also known as n1).
- 2) Vectors rv1 and rvo cannot be the same.

## **vramp**

*Generate a Ramp in a Vector*

### **C USAGE:**

```
float rvo[], rs1, rs2;  
long inco, n;  
vramp_(&rs1, &rs2, rvo, &inco, &n);  
-or-  
vramp(&rs1, &rs2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rvo(), rs1, rs2  
integer*4 inco, n  
call vramp(rol, rs2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rs1	(inp) - Initial ramp value
rs2	(inp) - Ramp increment
rvo	(out) - Real output vector
inco	(inp) - Increment for rvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Generates a linear ramp starting with rs1 with successive points differing by rs2. The ramp is written to output vector rvo.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rs1 + rs2*i, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vrand**

*Generate Random Number Vector*

### **C USAGE:**

```
float rvo[];  
long is1, inco, n;  
vrand_(&is1, rvo, &inco, &n);  
-or-  
vrand(&is1, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rvo()  
integer*4 is1, inco, n  
call vrand(is1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

is1	(i/o) - Input 32 bit seed, output last seed generated
rvo	(out) - Real output vector
inco	(inp) - Increment for rvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Generates a random vector using the seed from is1 based on a linear congruential method outputting numbers in the range from 0.0 to 1.0.

### **MATHEMATICAL EQUIVALENT:**

```
is1 = is1 * 1103515245 + 12345  
rvo[i] = (is & 0x007fffff) / POW{ 2, 23 }, for i = 0 to n-1
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vreal**

*Extract Real Part of Complex Vector*

### **C USAGE:**

```
float cv1[][2], rvo[];
long incl, inco, n;
vreal_(cv1, &incl, rvo, &inco, &n);
-or-
vreal(cv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
complex cv1()
real rvo()
integer*4 incl, inco, n
call vreal(cv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

cv1	(inp) - Complex input vector
incl	(inp) - Vector increment for cv1
rvo	(out) - Real output vector
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The real part of the complex input vector cv1 is copied to real output vector rvo.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco]=\text{REAL}\{ cv1[i*incl] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vrecip**

*Reciprocal of Vector*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vrecip_(rv1, &incl, rvo, &inco, &n);
-or-
vrecip(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vrecip(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment for rv1
rvo	(out) - Real output vector
inco	(inp) - Vector increment for rvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

The reciprocal of each point in the input vector is written to the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = 1.0/rv1[i*incl], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vrsqrt**

## *Vector Reciprocal Square Root*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vrsqrt_(rv1, &incl, rvo, &inco, &n);
-or-
vrsqrt(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vrsqrt(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the reciprocal of the square root of each element of the real input vector and places the result into the real output vector.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = 1.0 / \text{SQRT}\{rv1[i*incl]\}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

No checks are made for division by zero or negative square roots.

## **vrvrs**

## *Reverse a Vector*

### **C USAGE:**

```
float rv1[];  
long incl, n;  
vrvrs_(rv1, &incl, &n);  
-or-  
vrvrs(rv1, incl, n);
```

### **FORTRAN USAGE:**

```
real rv1()  
integer*4 incl, n  
call vrvrs(rv1, incl, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(i/o) - Real vector to reverse
incl	(inp) - Increment for vector
n	(inp) - Number of points to reverse

### **SUBROUTINE DESCRIPTION:**

Reverses the order of the desired points in a vector. (Points towards the beginning of the vector are swapped with points towards the end of the vector.)

### **MATHEMATICAL EQUIVALENT:**

$$rv1[i*incl] = rv1[((n-i-1)*incl)], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

The number of points must be greater than one.

## **vsadd**

*Add a Scalar to a Vector*

### **C USAGE:**

```
float rv1[], rs1, rvo[];
long incl, inco, n;
vsadd_(rv1, &incl, &rs1, rvo, &inco, &n);
-or
vsadd(rv1, incl, &rs1, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rs1, rvo()
integer*4 incl, inco, n
call vsadd(rv1, incl, rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rs1	(inp) - Real scalar to add
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Adds the scalar rs1 to the desired points in the input vector and stores the result in the output vector.

### **MATHEMATICAL EQUIVALENT:**

$rvo[i*inco] = rv1[i*incl] + rs1$ , for  $i=0$  to  $n-1$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **vsbm**

## *Vector Subtract and Multiply*

### **C USAGE:**

```
float rv1[], rv2[], rv3[], rvo[];
long inc1, inc2, inc3, inco, n;
vsbm_(rv1, &inc1, rv2, &inc2, rv3, &inc3, rvo, &inco, &n);
-or-
vsbm(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rv3(), rvo()
integer*4 inc1, inc2, inc3, inco, n
call vsbm(rv1, inc1, rv2, inc2, rv3, inc3, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rv3	(inp) - Real input vector 3
inc3	(inp) - Vector increment for rv3
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Subtracts real input vectors rv2 from rv1, then multiplies by input vector rv3, and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = (rv1[i*inc1] - rv2[i*inc2]) * rv3[i*inc3], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vsbsbm**

*Vector Subtract, Subtract, and Multiply*

### **C USAGE:**

```
float rv1[], rv2[], rv3[], rv4[], rvo[];
long inc1, inc2, inc3, inc4, inco, n;
vsbsbm_(rv1, &inc1, rv2, &inc2, rv3, &inc3, rv4, &inc4, rvo, &inco, &n);
-or-
vsbsbm(rv1, inc1, rv2, inc2, rv3, inc3, rv4, inc4, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rv3(), rv4(), rvo()
integer*4 inc1, inc2, inc3, inc4, inco, n
call vsbebm(rv1, inc1, rv2, inc2, rv3, inc3, rv4, inc4, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real vector one
inc1	(inp) - Increment for rv1
rv2	(inp) - Real vector two
inc2	(inp) - Increment for rv2
rv3	(inp) - Real vector three
inc3	(inp) - Increment for rv3
rv4	(inp) - Real vector four
inc4	(inp) - Increment for rv4
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Takes the difference of real input vectors rv1 and rv2, then multiplies by the difference of input vectors rv3 and rv4, and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = (rv1[i*inc1] - rv2[i*inc2]) * (rv3[i*inc3] - rv4[i*inc4]), \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vsbsm**

## *Vector Subtract and Scalar Multiply*

### **C USAGE:**

```
float rv1[], rv2[], rvo[], rs1;
long incl, inc2, inco, n;
vsbsm_(rv1, &incl, rv2, &inc2, &rs1, rvo, &inco, &n);
-or-
vsbsm(rv1, incl, rv2, inc2, &rs1, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo(), rs1
integer*4 incl, inc2, inco, n
call vsbsm(rv1, incl, rv2, inc2, rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real vector one
incl	(inp) - Increment for rv1
rv2	(inp) - Real vector two
inc2	(inp) - Increment for rv2
rs1	(inp) - Real scalar multiplier
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Subtracts the input real vector rv2 from real vector rv1, then multiplies by real scalar rs1, and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = (rv1[i*incl] - rv2[i*inc2]) * rs1, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vscatr**

## *Vector Scatter*

### **C USAGE:**

```
float rv1[], rvo[];
long iv2 [], inc1, inc2, n;
vscatr_(rv1, &inc1, iv2, &inc2, rvo, &n);
-or-
vocatr(rv1, inc1, iv2, inc2, rvo, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 iv2(), inc1, inc2, n
call vscatr(rv1, inc1, iv2, inc2, rvo, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
inc1	(inp) - Vector increment for rv1
iv2	(inp) - Integer input vector
inc2	(inp) - Vector increment for iv2
rvo	(out) - Real compact output vector
n	(inp) - Number of points to output

### **SUBROUTINE DESCRIPTION:**

Points from the input vector rv1 are placed into the output vector rvo at locations specified by the indices in the vector iv2. The index 0 specifies the first storage location in rvo.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[iv2[i*inc2]] = rv1[i*inc1], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

Note that locations in rvo, that are not the target of a data transfer will retain their previous values.

## **vsdiv**

*Divide Vector by Scalar*

### **C USAGE:**

```
float rv1[], rs1, rvo[];
long incl, inco, n;
vsdiv_(rv1, &incl, &rs1, rvo, &inco, &n);
-or-
vsdiv(rv1, incl, &rs1, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rs1, rvo()
integer*4 incl, inco, n
call vsdiv(rv1, incl, rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rs1	(inp) - Real divisor (non-zero)
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Divides the selected points in the input vector by the real scalar and stores the results to the output vector rvo.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*incl] / rs1, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

No explicit check is made for divide by zero.

## **vsimps**

## *Simpson's Rule Integration*

### **C USAGE:**

```
float rv1[], rvo[], rs1;
long incl, inco, n;
vsimps_(rv1, &incl, rvo, &inco, &n, &rs1);
-or-
vsimps(rv1, incl, rvo, inco, n, &rs1);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo(), rs1
integer*4 incl, inc2, n
call vsimps(rv1, incl, rvo, inco, n, rs1)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of points in output vector
rs1	(inp) - The integration step size

### **SUBROUTINE DESCRIPTION:**

Integrates the vector rv1 with step size rs1 and stores the result in vector rvo, using the Simpson's rule for integration.

### **MATHEMATICAL EQUIVALENT:**

$$\begin{aligned} rvo[0] &= 0.0 \\ rvo[inco] &= rs1/2.0 * (rv1[0] + rv1[incl]) \\ rvo[i*inco] &= rvo[(i-2)*inco] + rs1/3.0 * [rv1[(i-2)*incl] + 4.0*rv1[(i-1)*incl] + rv1[i*incl]] \end{aligned}$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

The input and output vector must be different.

## **vsin**

## *Vector Sine*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vsin_(rv1, &incl, rvo, &inco, &n);
-or-
vsin(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vsin(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the sine of each element of the real input vector and places the results in the real output vector. The elements of the input vector rv1 are assumed to be in radians.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{SIN} \{rv1[i*incl]\}$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vsinf**

## *Vector Sine in Fractions*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vsinf_(rv1, &incl, rvo, &inco, &n);
-or-
vsinf(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vsinf(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the sine of each element of the real input vector and places the result the real output vector. The elements of the input vector rv1 are assumed to be in fractions of a circle.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{SIN} \{rv1[i*incl] * 2\pi \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **vsinrf**

*Vector Sine in Fractions, Reduced Range*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vsinf_(rv1, &incl, rvo, &inco, &n);
-or-
vsinf(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vsinf(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the sine of each element of the real input vector and places the result the real output vector. The elements of the input vector rv1 are assumed to be in fractions of a circle in the range of -.25 to +.25.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{SIN} \{rv1[i*incl] * 2\pi \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

Results are unpredictable for input values outside of expected range.

## **vsm2sa**

### *Multiply Two Vectors by Scalars and Add a Scalar*

#### **C USAGE:**

```
float rv1[], rv2[], rs1, rs2, rs3, rvo[];
long inc1, inc2, inco, n;
vsm2sa_(rv1, &inc1, &rs1, rv2, &inc2, &rs2, &rs3, rvo, &inco, &n);
-or-
vsm2sa(rv1, inc1, &rs1, rv2, inc2, &rs2, &rs3, rvo, inco, n);
```

#### **FORTRAN USAGE:**

```
real rv1(), rv2(), rs1, rs2, rs3, rvo()
integer*4 inc1, inc2, inco, n
call vsm2sa(rv1, inc1, rs1, rv2, inc2, rs2, rs3, rvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rs1	(inp) - Scalar to multiply with rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rs2	(inp) - Scalar to multiply with rv2
rs3	(inp) - Scalar to add
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

For each point to be output, a sum is generated of: a point from the first input vector multiplied by its scalar, a point from the second input vector multiplied by its scalar, and finally the additive scalar. The result is written to the output vector.

#### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = (rv1[i*inc1] * rs1) + (rv2[i*inc2] * rs2) + rs3, \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vsma**

## *Vector Scalar Multiply and Add*

### **C USAGE:**

```
float rv1[], rv2[], rvo[], rs1;
long incl, inc2, inco, n;
vsma_(rv1, &incl, &rs1, rv2, &inc2, rvo, &inco, &n);
-or-
vsma(rv1, incl, &rs1, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo(), rs1
integer*4 incl, inc2, inco, n
call vema(rv1, incl, rs1, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment for rv1
rs1	(inp) - Real scalar multiplier
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Multiplies the input real vector rv1 by real scalar rs1, adds input real vector rv2, and puts the results in real output vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i * inco] = rv1[i * incl] * rs1 + rv2[i * inc2], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vsma2**

### *Two Vector Multiply and Scalar Add*

#### **C USAGE:**

```
float rv1[], rv2[], rvo[], rs1, rs2;
long inc, inco, n;
vsma2_(rv1, &rs1, rv2, &rs2, &inc, rvo, &inco, &n);
-or-
vsma2(rv1, &rs1, rv2, &rs2, inc, rvo, inco, n);
```

#### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 inc, inco, n
real rs1, rs2
call vsma2(rv1, rs1, rv2, rs2, inc, rvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
rs1	(inp) - Real scalar multiplicative factor
rv2	(inp) - Real input vector 2
rs2	(inp) - Real scalar multiplicative factor
inc	(inp) - Vector increment for rv1 and rv2
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

Multiplies real input vectors rv1 and rv2, both with the same increment inc, by real scalar inputs rs1 and rs2, and puts the result in output real vector rvo for n points.

#### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*inc] * rs1 + rv2[i*inc] * rs2, \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vsma3**

### *Three Vector Multiply and Scalar Add*

#### **C USAGE:**

```
float rv1[], rv2[], rv3[], rvo[], rs1, rs2, rs3;
long inc, inco, n;
vsma3_(rv1, &rs1, rv2, &rs2, rv3, &rs3, &inc, rvo, &inco, &n);
-or-
vsma3(rv1, &rs1, rv2, &rs2, rv3, &rs3, inc, rvo, inco, n);
```

#### **FORTRAN USAGE:**

```
real rv1(), rv2(), rv3(), rvo(), rs1, rs2, rs3
integer*4 inc, inco, n
call vsma3(rv1, rs1, rv2, rs2, rv3, rs3, inc, rvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
rs1	(inp) - Multiplier for rv1
rv2	(inp) - Real input vector 2
rs2	(inp) - Multiplier for rv2
rv3	(inp) - Real input vector 3
rs3	(inp) - Multiplier for rv3
inc	(inp) - Increment for rv1, rv2, rv3
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of points to move

#### **SUBROUTINE DESCRIPTION:**

Multiplies real input vectors rv1, rv2, and rv3, all with the same increment inc, by real scalar inputs rs1, rs2, and rs3, respectively, and puts the result in output real vector rvo for n points.

#### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*inc] * rs1 + rv2[i*inc] * rs2 + rv3[i*inc] * rs3, \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vsma4**

### *Four Vector Multiply and Scalar Add*

#### **C USAGE:**

```
float rv1[], rv2[], rv3[], rv4[], rvo[], rs1, rs2, rs3, rs4;
long inc, inco, n;
vsma4_(rv1, &rs1, rv2, &rs2, rv3, &rs3, rv4, &rs4, &inc, rvo, &inco, &n);
-or-
vsma4(rv1, &rs1, rv2, &rs2, rv3, &rs3, rv4, &rs4, inc, rvo, inco, n);
```

#### **FORTRAN USAGE:**

```
real rv1(), rv2(), rv3(), rv4(), rvo(), rs1, rs2, rs3, rs4
integer*4 inc, inco, n
call vsma4(rv1, rs1, rv2, rs2, rv3, rs3, rv4, rs4, inc, rvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
rs1	(inp) - Multiplier for rv1
rv2	(inp) - Real input vector 2
rs2	(inp) - Multiplier for rv2
rv3	(inp) - Real input vector 3
rs3	(inp) - Multiplier for rv3
rv4	(inp) - Real input vector 4
rs4	(inp) - Multiplier for rv4
inc	(inp) - Increment for rv1, rv2, rv3, rv4
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

Multiplies real input vectors rv1, rv2, rv3, and rv4, all with same increment inc, by real scalar inputs rs1, rs2, rs3, and rs4 respectively, and puts the result in output real vector rvo for n points.

#### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*inc] * rs1 + rv2[i*inc] * rs2 + rv3[i*inc] * rs3 + rv4[i*inc] * rs4, \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vsmsa**

*Multiply Vector by a Scalar and Add Scalar*

### **C USAGE:**

```
float rv1[], rs1, rs2, rvo[];
long incl, inco, n;
vsmsa_(rv1, &incl, &rs1, &rs2, rvo, &inco, &n);
-or-
vsmsa(rv1, incl, &rs1, &rs2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rs1, rs2, rvo()
integer*4 incl, inco, n
call vsmsa(rv1, incl, rs1, rs2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rs1	(inp) - Scalar to multiply
rs2	(inp) - Scalar to add
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Each selected point in the input vector is multiplied by the real scalar and then has the real scalar added to it. The result is stored in the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = (rv1[i*incl] * rs1) + rs2, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vsmsb**

## *Vector Scalar Multiply and Subtract*

### **C USAGE:**

```
float rv1[], rv2[], rvo[], rs1;
long incl, inc2, inco, n;
vsmsb_(rv1, &incl, &rs1, rv2, &inc2, rvo, &inco, &n);
-or-
vsmsb(rv1, incl, &rs1, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo(), rs1
integer*4 incl, inc2, inco, n
call vsmsb(rv1, incl, rs1, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
incl	(inp) - Vector increment for rv1
rs1	(inp) - Real scalar multiplier
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Multiplies the input real vector rv1 by real scalar rs1, subtracting input real vector rv2, and putting the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*incl] * rs1 - rv2[i*inc2], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## **vsmul**

*Multiply Vector by a Scalar*

### **C USAGE:**

```
float rv1[], rs1, rvo[];
long incl, inco, n;
vsmul_(rv1, &incl, &rs1, rvo, &inco, &n);
-or-
vsmul(rv1, incl, &rs1, rvo, inco, &n);
```

### **FORTRAN USAGE:**

```
real rv1(), rs1, rvo()
integer*4 incl, inco, n
call vsmul(rv1, incl, rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rs1	(inp) - Scalar to multiply
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Multiplies the selected points in the input vector by the real scalar and stores the result in the output vector.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*incl] * rs1, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vspdp**

## *Vector Convert Single to Double Precision*

### **C USAGE:**

```
float rv1[];  
double dvo[];  
long incl, inco, n;  
vspdp_(rv1, &incl, dvo, &inco, &n);  
-or-  
vspdp(rv1, incl, dvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1()  
real*8 dvo()  
integer*4 incl, inco, n  
call vspdp(rv1, incl, dvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
dvo	(out) - Double precision output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of points to convert

### **SUBROUTINE DESCRIPTION:**

Converts the input real vector rv1 to double precision and puts the result in output double precision vector dvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$\text{dvo}[i*\text{inco}] = \text{DOUBLE}\{ \text{rv1}[i*\text{incl}] \}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vsq**

## *Vector Square*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vsq_(rv1, &incl, rvo, &inco, &n);
-or-
vsq(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vsq(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment for rv1
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Squares real input vector rv1 and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*incl]^2, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vsqrt**

## *Vector Square Root*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vsqrt_(rv1, &incl, rvo, &inco, &n);
-or-
vsqrt(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vsqrt(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the square root of each element of the real input vector and places the result into the real output vector.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{SQRT}\{rv1[i*incl]\}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

No check is made for negative square roots.

## **vssq**

## *Vector Signed Square*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vssq_(rv1, &incl, rvo, &inco, &n);
-or-
vssq(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n call
vssq(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment for rv1
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Multiplies the real input vector rv1 with its absolute value and puts the result in output real vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv1[i*incl] * |rv1[i*incl]|, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vsub**

*Subtract One Vector from Another*

### **C USAGE:**

```
float rv1[], rv2[], rvo[];
long inc1, inc2, inco, n;
vsub_(rv1, &inc1, rv2, &inc2, rvo, &inco, &n);
-or-
vsub(rv1, inc1, rv2, inc2, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo()
integer*4 inc1, inc2, inco, n
call vsub(rv1, inc1, rv2, inc2, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector 1
inc1	(inp) - Vector increment for rv1
rv2	(inp) - Real input vector 2
inc2	(inp) - Vector increment for rv2
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Subtracts vector rv2's points from vector rv1 and stores the results in the output vector rvo.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = rv2[i*inc2] - rv1[i*inc1], \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vsum**

## *Vector Sum*

### **C USAGE:**

```
float rv1[], rvo[], rs1;
long incl, inco, n;
vsum_(rv1, &incl, rvo, &inco, &n, &rs1);
-or-
vsum(rv1, incl, rvo, inco, n, &rs1);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo(), rs1
integer*4 incl, n call
vsum(rv1, incl, rvo, inco, n, rs1)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment for rv1
rvo	(out) - Real output vector
inco	(inp) - Vector increment for output vector
n	(inp) - Number of points in output vector
rs1	(inp) - Integration step size

### **SUBROUTINE DESCRIPTION:**

The values in vector rv1 are integrated using the step size rs1. The resulting integral vector is stored into rvo.

### **MATHEMATICAL EQUIVALENT:**

```
rvo[0] = rv1[0] * rs1
rvo[i*inco] = rvo[(i-1)*inco] + rs1*rv1[i*incl], for i=1 to n-1
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vswap**

## *Vector Swap*

### **C USAGE:**

```
float rv1[], rv2[];
long inc1, inc2, n;
vswap_(rv1, &inc1, rv2, &inc2, &n);
-or-
vswap(rv1, inc1, rv2, inc2, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rv2()
integer*4 inc1, inc2, n
call vswap(rv1, inc1, rv2, inc2, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(i/o) - Real vector 1
inc1	(inp) - Vector increment for rv1
rv2	(i/o) - Real vector 2
inc2	(inp) - Vector increment for rv2
n	(inp) - Number of points to swap

### **SUBROUTINE DESCRIPTION:**

The elements of vectors rv1 and rv2 are swapped with each other.

### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    rv2[i*inc2] = rv1[i*inc1]
    rv1[i*inc1] = rv2[i*inc2]
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

Note that elements skipped due to non-compact increments retain their original values.



## **vtabi**

### *Vector Table Look-up, Linear Interpolate*

#### **C USAGE:**

```
float rv1[], rv2[], rvo[], rs1, rs2;
long incl, n2, inco, n;
vtabi_(rv1, &incl, &rs1, &rs2, rv2, rvo, &inco, &n2, &n);
-or-
vtabi(rv1, incl, &rs1, &rs2, rv2, rvo, inco, n2, n);
```

#### **FORTRAN USAGE:**

```
real rv1(), rv2(), rvo(), rs1, rs2
integer*4 incl, n2, inco, n
call vtabi(rv1, incl, rs1, rs2, rv2, rvo, inco, n2, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Integer increment for rv1
rs1	(inp) - Real multiplicative scalar
rs2	(inp) - Real additive scalar
rv2	(inp) - Second real input vector (table)
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n2	(inp) - Integer size of vector (table)
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

After scaling points in rv1, table look-up and linear interpolation are used with table rv2 to determine the value that will be written into rvo.

#### **MATHEMATICAL EQUIVALENT:**

```
tmp = rs 1 * rv1[i*incl] + rs2
tmp = MAX{ 1, MIN{ n2, tmp} }
idx = INT{ tmp }
fac = tmp - idx
rvo = rv2[idx] + fac * (rv2[idx+1] - rv2[idx])
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vtan**

## *Vector Tangent*

### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vtan_(rv1, &incl, rvo, &inco, &n);
-or-
vtan(rv1, incl, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vtan(rv1, incl, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Computes the tangent of each element of the real input vector and places the results in the real output vector. The elements of the input vector are assumed to be in radians.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco]=TAN\{rv1[i*incl]\}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

No check is made for invalid input to TAN.

## **vtanf**

### *Vector Tangent in Fractions of a Circle*

#### **C USAGE:**

```
float rv1[], rvo[];
long incl, inco, n;
vtanf_(rv1, &incl, rvo, &inco, &n);
-or-
vtanf(rv1, incl, rvo, inco, n);
```

#### **FORTRAN USAGE:**

```
real rv1(), rvo()
integer*4 incl, inco, n
call vtanf(rv1, incl, rvo, inco, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Increment for input vector
rvo	(out) - Real output vector
inco	(inp) - Increment for output vector
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

Computes the tangent of each element in the real input vector and places the result in the real output vector. The elements of the input vector are assumed to be in fractions of a circle.

#### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{TAN} \{rv1[i*incl] * 2\pi \}, \text{ for } i=0 \text{ to } n-1$$

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vthr**

## *Vector Threshold*

### **C USAGE:**

```
float rv1[], rvo[], rs1;  
long incl, inco, n;  
vthr_(rv1, &incl, &rs1, rvo, &inco, &n);  
-or-  
vthr(rv1, incl, &rs1, rvo, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo(), rs1  
integer*4 incl, inco, n  
call vthr(rv1, incl, rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment for rv1
rs1	(inp) - Real scalar threshold
rvo	(out) - Real vector output
inco	(out) - Vector increment for rvo
n	(inp) - Number of samples to process

### **SUBROUTINE DESCRIPTION:**

Compare vector points against threshold, take larger value and put into output vector rvo for n points.

### **MATHEMATICAL EQUIVALENT:**

$$rvo[i*inco] = \text{MAX}\{rv1[i*incl], rs1\}, \text{ for } i=0 \text{ to } n-1$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vthres**

*Replace Elements Less than Scalar with Zero*

### **C USAGE:**

```
float rv1[], rvo[], rs1;
long incl, inco, n;
vthres_(rv1, &incl, &rs1, rvo, &inco, &n);
-or-
vthres(rv1, incl, &rs1, rvo,, inco, n);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo(), rs1
integer*4 incl, inco, n
call vthres(rv1, incl, rs1, rvo, inco, n)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment for rv1
rs1	(inp) - Real scalar minimum value
rvo	(out) - Real vector output
inco	(inp) - Vector increment for rvo
n	(inp) - Number of points in output vector

### **SUBROUTINE DESCRIPTION:**

Copies elements from rv1 into rvo, substituting 0 if the element from rv1 is less than scalar rs1.

### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    if rv1[i*inc 1] < rs1 then rvo[i*inco] = 0
    else rvo[i*inco] = rv1[i*inc1]
```

### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vtrapz**

## *Trapezoidal Rule Integration*

### **C USAGE:**

```
float rv1[], rvo[], rs1;
long incl, inco, n;
vtrapz_(rv1, &incl, rvo, &inco, &n, &rs1);
-or-
vtrapz(rv1, incl, rvo, inco, n, &rs1);
```

### **FORTRAN USAGE:**

```
real rv1(), rvo(), rs1
integer*4 incl, inc2, n
call vtrapz(rv1, incl, rvo, inco, n, rs1)
```

### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
incl	(inp) - Vector increment for rv1
rvo	(out) - Real output vector
inco	(inp) - Output vector increment
n	(inp) - Number of samples to process
rs1	(inp) - The integration step size

### **SUBROUTINE DESCRIPTION:**

vtrapz integrates the vector rv1 with step size rs1 and stores the result in vector rvo, using the trapezoidal rule for integration.

### **MATHEMATICAL EQUIVALENT:**

$$\begin{aligned} rvo[0] &= 0.0 \\ rvo[i*inco] &= rvo[(i-1)*inco] + rs1/2.0 * (rv1[(i-1)*incl] + rv1[i*incl]), \text{ for } i=1, n-1 \end{aligned}$$

### **RESTRICTIONS & SPECIAL CONDITIONS:**

The input and output vector must be different.

## **VXCS**

### *Real Vector Multiplied by Complex Exponential*

#### **C USAGE:**

```
float rv1[], cvo[][2], rs1, rs2;
long inco, n;
vxcs_(rv1, cvo, &inco, &rs1, &rs2, &n);
-or-
vxcs(rv1, cvo, inco, &rs1, &rs2, n);
```

#### **FORTRAN USAGE:**

```
real*4 rv1(), rs1, rs2
complex cvo()
integer*4 inco, n
call vxcs(rv1, cvo, inco, rs1, rs2, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Real input vector
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for cvo
rs1	(inp) - Frequency
rs2	(i/o) - Phase
n	(inp) - Number of points in output vector

#### **SUBROUTINE DESCRIPTION:**

A complex exponential is formed from the initial phase rs2 and frequency rs1 (in units of radians). The real part of each element of the complex exponential is the cosine of the current phase step; the imaginary part is the sine of the current phase step. The final phase step is written to the scalar rs2. Each element of the input vector rv1 is multiplied by the current complex exponential to form the complex output vector cvo.

#### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    cvo[i*inco] = (rv1[i]* COS { i*rs1+rs2}, SIN{i*rs1+rs2} )
rs2 = n*rs1 + rs2
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None

## **vxcsf**

### *Real Vector Multiplied by Fractional Complex Exponential*

#### **C USAGE:**

```
float rv1[], cvo[][2], rs1, rs2;
long incl, inco, n;
vxcsf_(rv1, &incl, cvo, &inco, &rs1, &rs2, &n);
-or-
vxcsf(rv1, incl, cvo, inco, &rs1, &rs2, n);
```

#### **FORTRAN USAGE:**

```
real*4 rv1(), rs1, rs2
complex cvo()
integer*4 incl, inco, n
call vxcsf(rv1, incl, cvo, inco, rs1, rs2, n)
```

#### **ARGUMENT DESCRIPTIONS:**

rv1	(inp) - Complex input vector
incl	(inp) - Vector increment for rv1
cvo	(out) - Complex output vector
inco	(inp) - Vector increment for rvo
rs1	(inp) - Frequency
rs2	(i/o) - Phase
n	(inp) - Number of samples to process

#### **SUBROUTINE DESCRIPTION:**

A complex exponential is formed from the initial phase rs2 and frequency rs 1 (in units of fractions of a circle). The real part of each element of the complex exponential is the cosine of the current phase step; the imaginary part is the sine of the current phase step. The final phase step is written to the scalar rs2. Each element of the input vector rv1 is multiplied by the current complex exponential to form the complex output vector cvo.

#### **MATHEMATICAL EQUIVALENT:**

```
for i=0 to n-1:
    cvo[i*inco] = (rv1[i]* COS { 2π*(i*rs1+rs2)}, SIN {2π *(i*rs1+rs2)} )
rs2 = n*rs1 + rs2
```

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

None



## wiener

## Wiener-Levinson Equation Solution

### C USAGE:

```
float  R[], G[], F[], A[];
long  IERR, LR, ISW;
wiener_(&LR, R, G, F, A, &ISW, &IERR);
-or-
wiener(LR, R, G, F, A, ISW, &IERR);
```

### FORTRAN USAGE:

```
real*4  R(1), G(1), F(1), A(1)
integer*4 IERR, LR, ISW
call wiener(LR, R, G, F, A, ISW, IERR)
```

### ARGUMENT DESCRIPTIONS:

LR	(inp) - Length of the filter
R	(inp) - Real vector with Auto-correlation coefficients
G	(inp) - Real vector with Cross-correlation coefficients
ISW	(inp) - Mode selector switch ISW $\neq$ 0: General deconvolution, ISW = 0: Spike deconvolution.
F	(out) - Real array of filter coefficients.
A	(out) - Real array of coefficients of the prediction error operator.
IERR	(out) - Scalar Error flag. IERR = 0 The routine completed without error. IERR = -1 Invalid parameter, LR < 1. IERR = L The routine encountered a division by zero on the L-th pass of algorithm.

### SUBROUTINE DESCRIPTION:

To solve a system of single channel normal equations which arise in least-squares filtering and prediction problems.

### MATHEMATICAL EQUIVALENT:

WIENER solves the following set of N equations for F:

$\text{SUM} \{ F[i] * R[j-i] \} = G[j]; i = 0 \text{ to } n-1; j = 0 \text{ to } n-1$   
where  $R[-i] = R[i]$ , for  $i = 0 \text{ to } n-1$ .

WIENER solves the following set of N equations for A:

$\text{SUM} \{ A[i] * R[j-i] \} = V * d[j]; i = 0 \text{ to } n-1; j = 0 \text{ to } n-1$   
where  $A[0] = 1.0$ ,  $V = \text{SUM} \{ A[i] * R[i] \}$  for  $i = 0 \text{ to } n-1$ , and  
 $d[j] = 1$  when  $j = 0$  and  $d[j] = 0$  when  $j \neq 0$ .

WIENER solves a system of N equations of auto-correlation coefficients and cross-correlation coefficients for N filter-weighting coefficients and N prediction error operator coefficients. The auto-correlation coefficients are in input vector R and the cross-correlation coefficients are in input vector G. WIENER stores the filter-weighting coefficients in output vector F and the coefficients of the prediction error operator in output vector

A.

If the system of equations is solved, the error flag, IERR, is set to 0. If the system of equations cannot be solved, IERR is set to L, where L is the pass on which the failure occurred.

References:

E. A. Robinson, "Multichannel Time Series Analysis with Digital Computer Programs", Holden Day, 1967, pp. 43-46.

E. A. Robinson et al., "Digital Seismic Inverse Methods", International Human Resources Development Corporation, 1983, pp. 32-45.

Example:

LR = 5

ISW = 1

R : 10.00 4.00 -1.00 -4.00 -4.00

G : 4.00 2.00 0.00 -1.00 -1.00

Output:

F : 0.42 0.06 0.01 0.04 0.08

A : 1.00 -0.38 0.18 0.22 0.18

IERR = 0

#### **RESTRICTIONS & SPECIAL CONDITIONS:**

- 1) Neither F nor A can be overlayed.
- 2) Filter length LR must be less than 1023.

## Overview

The USPL can be used with numerous compilers and languages, it has native support for Borland C/C++, Borland C++ Builder and Microsoft Visual C++. Most any language that allows you to use a DLL will work with the USPL, as the DLL and DEF files are shipped with the standard version. The general steps to use the DLL are the same for all languages and compilers.

- 1) #define USPL\_DLL\_IMPORTS either before including header file or via compiler flags.
- 2) Include the header file in your source code
- 3) Make calls to the USPL per the API definitions
- 4) Include the appropriate USPL lib in your project or makefile
- 5) Compile/Link your code
- 6) Make sure the USPL.DLL and CW3230.DLL (memory manager) is available to your executable, either in the same directory, or in \Windows\System

## **Borland C++/C++ Builder**

Borland C++ 5.x, C++ Builder 1, and C++ Builder 3 are directly supported by USPL. The correct header file is “uspl.h” and the library to link to is uspl.lib. The sample code will compile, using the included xxxx\_bc5.ide for BC 5.x, or xxxx\_bcb.mak for BCB 1.0, or xxxx\_bcb3.bpr for BCB 3.0.

Unfortunately there is no environmental variable for Borland C/C++ 5.x, for the include and library directories of the compiler. Therefore the .mak file will need to be modified to point to the installed directories for include and lib.

## **Microsoft Visual C++**

Microsoft Visual C++ is directly supported by USPL. The correct header file is "uspl.h" and the library to link to is uspl\_vc.lib. The sample code will compile, using the included xxxxx\_vc.dsp. If for any reason you wish to rebuild the import library, use the uspl\_vc.def file for module definitions.

## **Borland Delphi**

Functions in the USPL may be used with Delphi, however the user must write their own header files for the routines that they want to use. All function calls are implemented using the C calling mechanism. Examine "uspl.h" for examples of calling conventions.

## Other Compilers

Any C/C++ compiler that has difficulty with the shipped libraries can be used if you have the source code version of the USPL, as you can simply recompile the code. If you do not have the source code version, and the compiler or language you are using does not support provided the .lib files, the .DEF file can be used to create a new .lib file for your compiler.

Sigma Tech is always working to port our products to other hardware platforms and compilers, so please contact us to see if a version is available for your platform and compiler, and if not if a port would be feasible.

## Example Programs

There are three example programs included with the USPL. Each example has make files for BC 5.x, BCB 1.0, BCB 3.0, and VC 6.0 using the naming convention explained under compiler usage. Each example is in a separate directory, along with executables created from each compiler. For maximum compatibility between compilers, most examples are Win32 console applications that can be run in DOS box.

The first sample is called “audio example” that loads a .WAV file from disk, allows the user to add reverb by supplying a delay and amplitude, and saves the file out as .WAV. This example demonstrates, using several basic USPL routines for data conversion, basic vector arithmetic, muxing and demuxing.

The second sample is called “fft example” that creates a sine wave in memory based on user parameters, fft's the data, finds the frequency of the sine wave, and performs an inverse fft and compares it to the original signal. This example illustrates vector generation, frequency domain processing, and vector search routines.

The final example is more of a utility than an example. This program computes timings for the USPL routines and compares them to timings done on other machines. This will allow you to compare the performance of your computer to other systems.



## Audio Example

This example program is compiled using the `audio_xxx.xxx` as appropriate for the compiler you are using. The USPL directory containing “`uspl.h`” must be set in the project as well as where `uspl_xx.lib` is located.

This example uses a class called `AudioC` to load/save .WAV files and the USPL for signal processing. The program loads a specified .wav, applies a reverb based on the user parameters, then saves the resulting .WAV file. The user provides input filename, output filename, delay and amplitude on the command line as follows:

```
audio_bcb input_file output_file delay amplitude
```

A sample .wav file “`chord.wav`” is included as well as a sample output “`result.wav`”. This was generated using the following command line:

```
audio_bcb chord.wav result.wav 35 50
```

The Audio Class code is contained in `AudioC.cpp`, the main signal processing code is in `reverb.cpp`. The source code is well commented and should be self-explanatory. This program only supports uncompressed PCM wav files.

## FFT Example

This example program is compiled using the `fft_xxx.xxx` as appropriate for the compiler you are using. The USPL directory containing “`uspl.h`” must be set in the project as well as where `uspl_xx.lib` is located.

The program generates a sine wave with a user specified frequency and amplitude. This signal is then FFT'd and the maximum frequency bin found. The maximum frequency is then computed and displayed for the user. The frequency displayed should be close to the specified sine frequency, dependant on the `fft_size`. The user provides input parameters on the command line as follows:

```
fft_vc sine_freq sine_amplitude fft_size
```

The main program is in `main.c`, and the signal processing code is contained in `sigproc.c`. This example is straight C code that has been commented and should be self-explanatory for users that understand frequency domain signal processing.

## Timer Example

This example is a utility that measures how long it takes to execute various functions in the USPL. The functions are compared to timings on two other computers showing how the performance of CPU stacks up. This routine attempts to calculate the overhead involved with each call, but it is important to note that the overhead value is only valid for routines that have a linear relationship between the number of points executed and execution time. This means that overhead numbers that are negative, etc. for such functions as FFTs, etc. are ok. The overhead cannot be measured with the sampling method employed by the program.

The executable for the timer is included so you can measure times on your machines. The executable should be run in a maximized dos box. All extraneous programs should be terminated in windows. The only programs that should be running is Explorer, and Systray, and the DOS box. Run the timer using the following syntax:

```
timer_bc5 > results
```

The column of interest are Sigma meas(ms/1K) which indicates how many milliseconds it took to run the routine on an input vector of 1024 points. The only variation from this is the FFT functions which have the function name followed by the fft size, ex. CFFTF\_16 is a 16 point CFFTF. The SC-2 and SC-3 numbers are for comparison purposes showing your performance versus i860 signal processing chips clocked at 40 and 50Mhz respectively.

This will run the timer program and save the results to a text file, we are always looking for timings from various machines and configurations. Email us your results if you want at: [uspl@sigmatechcorp.com](mailto:uspl@sigmatechcorp.com).

NOTE: Timing results using the EVALUATION version are much lower than the REGISTERED version. This is for several reasons: 1) The evaluation version makes checks during each call to see if it should display the reminder, this slows down performance. 2) The registered version has many optimizations that dramatically increase performance.

## Evaluation Version

The evaluation version of USPL is licensed for you to use for up to 30 days, at which time you must discontinue using the library or purchased the registered version of the software as dictated by the software license found in license.txt.

The published USPL routines are the same in both the evaluation and registered version of the library, and your source code can be compiled with either without any code changes. In the evaluation version, a popup reminder will be displayed every so often and program execution will halt until OK is pressed. This is to help encourage people to use the registered version of the software instead of using the evaluation version ad infinitum.

The most significant difference is that many routines have been optimized, and the optimized versions are only available in the registered version. Some of the optimized routines execute up to 4 times faster, thus allowing you to get the maximum performance from your application.

Questions and information regarding purchasing the registered version can be found the REGISTER.TXT file included with the distribution.

## **Registered Version**

The registered version of the USPL is subject to the licensing agreement found in license.txt. You may distribute the USPL.DLL and CW3230.DLL with your application to end users, but you are not allowed to distribute any other files from the registered version.

The registered version has highly optimized versions of many of the routines, which execute up to 4 times faster than the evaluation version. To switch between the registered and evaluation versions, just recompile your code, the API is the same for both.

By registering ordering the registered version, you will get the latest version of the software, and be included on email mailing list for updates. Questions and information regarding purchasing the registered version can be found the REGISTER.TXT file included with the distribution.

## **Installation**

For Win32 platforms simply unzip the package in a directory of your choosing. Make sure that you allow your decompression program to create the stored directories in order to preserve the file structure. If you don't the make files will not operate correctly.

Note: You will have to change the makefiles for the Borland C/C++ 5.x to point to the installed Borland directory for the include and library files.

