UCRL-ID-125968 Rev 1

IDP++: Signal and Image Processing Algorithms in C++ version 4.1

S. K. Lehman

November 8, 1996

Lawrence Livermore National Laboratory

This is an informal report intended primarily for internal or limited external distribution. The opinions and conclusions stated are those of the author and may or may not be those of the Laboratory.
Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This report has been reproduced
directly from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (615) 576-8401, FTS 626-8401

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161

Abstract

IDP++ (Image and Data Processing in C++) is a collection of signal and image processing algorithms written in C++. It is a compiled signal processing environment which supports four data types of up to four dimensions. It is developed within Lawrence Livermore National Laboratory's Image and Data Processing group as a partial replacement for View [3]. IDP++ takes advantage of the latest, implemented and actually working, object-oriented compiler technology to provide "information hiding." Users need only know C, not C++. Signals are treated like any other variable with a defined set of operators and functions in an intuitive manner. IDP++ is designed for real-time environment where interpreted signal processing packages are less efficient.

IDP++ exists for both SUNs and Silicon Graphics using their most current compilers.

Contents

1 Changes and New Functions 8
1.1 Changes to Existing Functions Since Version 3.5 . . . 8
1.2 New Functions and Features . . . 8

2 Introduction 10

3 Installing IDP++ 11

4 Writing, Compiling, and Linking 12

5 Notation Conventions 13

6 Signal Class 14
6.1 Properties . . . 14
6.2 Constructors . . . 15
6.2.1 Default Constructor . . . 15
6.2.2 Copy Constructor . . . 17
6.2.3 Detailed Constructor . . . 18
6.2.4 Size Constructor . . . 21
6.2.5 File Constructor . . . 23
6.2.6 Mixed Constructor . . . 24
6.3 Assignment Operators . . . 25
6.3.1 Signal to Signal Assignment Operators . . . 25
6.3.2 File to Signal Assignment Operators . . . 26
6.3.3 Scalar to Signal Assignment Operator . . . 27
6.4 Assignment Arithmetic Operators . . . 28
6.4.1 Signal to Signal Assignment Arithmetic Operators . . . 28
6.4.2 Scalar to Signal Assignment Arithmetic Operators . . . 28
6.5 Binary Operators . . . 29
6.5.1 Signal-Signal Binary Operators . . . 29
6.5.2 Signal-Scalar Binary Operators . . . 29
6.6 Miscellaneous Signal Functions and Operators . . . 30
6.6.1 Signal Element Access . . . 30
6.6.2 Signal Casts . . . 30
6.6.3 Signal Unary Negation . . . 31
6.6.4 Number of Elements . . . 31
6.6.5 Signal Dimensionality Access . . . 31
6.6.6 Signal Dimension Size Access . . . 32
6.6.7 Signal Origin Access . . . 32
6.6.8 Signal Interval Access . . . 33
6.6.9 Signal Data Member Access . . . 34
6.6.10 Signal Name Member Access . . . 34
6.6.11 Signal Directory Member Access . . . 35
6.6.12 Signal Type Member Access . . . 35
6.6.13 Signal Type Member Access . . . 36
6.6.14 Signal Minimum . . . 36
6.6.15 Signal Maximum . . . 36
6.6.16 Signal Sum . . . 37
6.6.17 Signal Constant Fill . . . 37
6.6.18 Saving Signals . . . 38
6.6.19 Appending Comments to SPR File . . . 38
6.6.20 Deleting a Signal . . . 39

6.6.21 Listing Signals in ASCII . . . 40
6.6.22 Signal Show . . . 42
6.6.23 Sorting Signal Elements . . . 44
6.7 Get/Set a Point Within a Signal . . . 46
6.8 Get Rows and Columns of 2D Signals . . . 47
6.8.1 Signal to Signal Get Column . . . 47
6.8.2 Signal to Memory Get Column . . . 47
6.8.3 Signal to Signal Get Row . . . 47
6.8.4 Signal to Memory Get Row . . . 48
6.9 Store Rows and Columns of 2D Signals . . . 49
6.9.1 Column Storing . . . 49
6.9.2 Row Storing . . . 49
6.10 Signal Math Functions . . . 50
6.10.1 Phase of a Complex Signal . . . 50
6.10.2 Complex Conjugate . . . 50
6.10.3 Cosine . . . 50
6.10.4 Complex Exponential . . . 51
6.10.5 Exponential . . . 51
6.10.6 Magnitude . . . 51
6.10.7 Magnitude and Phase to Real and Imaginary Conversion . . . 52
6.10.8 Intensity . . . 52
6.10.9 Imaginary Part of a Complex Signal . . . 52
6.10.10 Logarithms . . . 53
6.10.11 Signal to a Power . . . 53
6.10.12 Real Part of a Complex Signal . . . 53
6.10.13 Sine . . . 54
6.10.14 Square Root . . . 54

7 Signal Class Functions **55**
7.1 Autocorrelate . . . 55
7.2 Average . . . 55
7.3 Barrel Shift . . . 56
7.4 Clip . . . 56
7.5 Central Moments . . . 57
7.6 Center of Mass . . . 59
7.7 Convolve . . . 59
7.8 Correlate . . . 59
7.9 Crosscorrelate . . . 60
7.10 Demean . . . 60
7.11 Detrend . . . 60
7.12 Differentiate . . . 61
7.13 Extract . . . 62
7.14 Histogram . . . 63
7.15 Insert . . . 63
7.16 Integrate . . . 64
7.17 Local Mean . . . 65
7.18 Local Statistics . . . 67
7.19 Magnify . . . 68
7.20 Merge . . . 68
7.21 First Two Moments of a Signal . . . 69
7.22 Flatten a Signal to 1 Dimension . . . 70
7.23 Fold a 1 Dimensional Signal to an N-Dimensional Signal . . . 70
7.24 Normalize . . . 71

7.25 Higher Order Approximation to First Derivative . . . 73
7.26 Overlay . . . 73
7.27 Probability Density Function . . . 74
7.27.1 PDF . . . 75
7.27.2 Interpolated PDF . . . 75
7.27.3 PDF from Histogram . . . 76
7.28 Powerspectrum . . . 79
7.29 Project . . . 79
7.30 Replicate . . . 79
7.31 Resample . . . 80
7.32 Rotate . . . 80
7.33 Shift . . . 81
7.34 Smooth . . . 81
7.35 Sum Magnitudes and Intensities of Signal Elements . . . 82
7.36 SNR . . . 83
7.37 Spectrogram . . . 85
7.38 Statistics . . . 86
7.39 Threshold . . . 88
7.40 Fourier Transforms . . . 89
7.40.1 Column Fourier Transforms . . . 91
7.40.2 Half-Plane Column Fourier Transforms . . . 93
7.40.3 Fourier Transform . . . 94
7.40.4 Half-Plane Fourier Transforms . . . 96
7.40.5 Row Fourier Transforms . . . 97
7.40.6 Half-Plane Row Fourier Transforms . . . 99
7.41 Order Statistic Filters . . . 101
7.41.1 α-Trimmed Mean . . . 101
7.41.2 Edge-Trimmed Mean . . . 102
7.41.3 MAX Filter . . . 102
7.41.4 Median Filter . . . 103
7.41.5 MIN Filter . . . 103
7.42 Signal Padding . . . 105
7.42.1 Center Padding . . . 105
7.42.2 Wrap-Around Padding . . . 105
7.42.3 Zero Padding . . . 106
7.43 Spline Interpolation . . . 107
7.43.1 Evaluate Spline . . . 108
7.43.2 Interpolate . . . 110
7.43.3 Computation of Spline Coefficients . . . 111
7.44 Random Number Generators . . . 113
7.44.1 Class A Random Numbers . . . 113
7.44.2 Exponentially Distributed Random Numbers . . . 114
7.44.3 Gamma Random Numbers . . . 115
7.44.4 Gaussian Random Numbers . . . 117
7.44.5 K-Distributed Random Numbers . . . 118
7.44.6 Uniform Random Numbers . . . 120
7.45 Matrix Operations . . . 121
7.45.1 Identity Matrix . . . 121
7.45.2 Matrix Inverse . . . 122
7.45.3 Matrix Equation Solver . . . 122
7.45.4 Vector Inner Product . . . 123
7.45.5 Euclidean Norm . . . 123
7.45.6 Determinant . . . 124

7.45.7 GAXMY . . . 124
7.45.8 GAXPY . . . 124
7.45.9 matrixform . . . 125
7.45.10 Matrix-Matrix Multiply . . . 126
7.45.11 Transpose Functions . . . 126
7.45.12 Weighted Inner Product . . . 127
7.46 Function Generators . . . 129
7.46.1 Bartlett Window . . . 129
7.46.2 Blackman Window . . . 130
7.46.3 Hamming Window . . . 131
7.46.4 Hanning Window . . . 132
7.46.5 Make a Box . . . 133
7.46.6 Make a Chirp . . . 134
7.46.7 Make a Circle or Sphere . . . 135
7.46.8 Make a Cosine . . . 136
7.46.9 Make an Ellipse or Oblate Spheroid . . . 137
7.46.10 Make a Complex Wave . . . 138
7.46.11 Make a Function . . . 139
7.46.12 Make a Gaussian . . . 140
7.46.13 Make a Ramp . . . 141
7.46.14 Make a Sine . . . 142
7.46.15 Make a Square Wave . . . 143
7.46.16 Make a Filter Window . . . 144

8 Array Class **145**
8.1 Constructors . . . 145
8.1.1 Default Constructor . . . 145
8.1.2 Copy Constructor . . . 145
8.1.3 Element Constructors . . . 146
8.2 Assignment Operators . . . 146
8.2.1 Array to Array Assignment . . . 146
8.2.2 Scalar to Array Assignment . . . 147
8.3 Array Element Access . . . 147
8.4 Array Casts . . . 148
8.5 Unary Negation . . . 148
8.6 Assignment Arithmetic Operators . . . 149
8.6.1 Array-Array Assignment Arithmetic Operators . . . 149
8.6.2 Scalar-Array Assignment Arithmetic Operators . . . 149
8.7 Binary Operators . . . 150
8.7.1 Array-Array Binary Operators . . . 150
8.7.2 Array-Scalar Binary Operators . . . 150
8.8 Array Relational Operators . . . 151
8.9 Elements in an Array . . . 151
8.10 Stream Operator . . . 151

9 Complex Class **153**
9.1 Constructors . . . 153
9.1.1 Default Constructor . . . 153
9.1.2 Data Constructor . . . 154
9.1.3 Copy Constructor . . . 154
9.2 Assignment Operators . . . 155
9.2.1 Complex to Complex Assignment . . . 155
9.2.2 Scalar to Complex Assignment . . . 155
9.3 Assignment Arithmetic Operators . . . 156

9.3.1 Complex to Complex Assignment Arithmetic Operators . . . 156
9.3.2 Scalar to Complex Assignment Arithmetic Operators . . . 156
9.4 Binary Operators . . . 157
9.4.1 Complex-Complex Binary Operators . . . 157
9.4.2 Complex-Scalar Binary Operators . . . 157
9.5 Extract Real and Imaginary Parts . . . 158
9.6 Casting . . . 159
9.7 Unary Negation . . . 159
9.8 Relational Operators . . . 160
9.9 Math Functions . . . 161
9.9.1 Phase of a Complex Variable . . . 161
9.9.2 Cube Root . . . 161
9.9.3 Complex Conjugate . . . 162
9.9.4 Cosine . . . 162
9.9.5 Euler Exponential . . . 163
9.9.6 Exponential . . . 165
9.9.7 Magnitude . . . 165
9.9.8 Intensity . . . 166
9.9.9 Natural Logarithm . . . 166
9.9.10 Base 10 Logarithm . . . 166
9.9.11 Convert from Magnitude and Phase to Real and Imaginary . . . 167
9.9.12 Convert to Polar Form . . . 167
9.9.13 Power . . . 168
9.9.14 Convert to Rectangular Form . . . 168
9.9.15 Return Rectangular Parts . . . 169
9.9.16 Sine . . . 169
9.9.17 Square Root . . . 170
9.10 Stream Operators . . . 170
9.11 Show . . . 171
9.12 Complex Class Type Definitions . . . 171

10 String Class **173**
10.1 Constructors . . . 173
10.1.1 Default Constructor . . . 173
10.1.2 Copy Constructor . . . 173
10.1.3 Constructor from a Character Array . . . 173
10.1.4 Constructor from a Letter . . . 174
10.1.5 Constructor from an Integer . . . 174
10.2 Casts . . . 174
10.3 Assignment Operators . . . 175
10.4 Assignment Arithmetic Operators . . . 175
10.4.1 Assignment Addition . . . 175
10.4.2 Assignment Subtraction . . . 175
10.5 Binary Operators . . . 176
10.5.1 String-String Binary Operators . . . 176
10.5.2 Char-String Binary Operators . . . 176
10.5.3 String-Char Binary Operators . . . 176
10.6 Relational Operators . . . 177
10.7 Show String Contents . . . 177
10.8 Parse a UNIX Path . . . 178
10.8.1 Get UNIX Path Base . . . 178
10.8.2 Get UNIX Path Directory . . . 178

11 View I/O Support Functions **179**
11.1 Append Text to SPR File . . . 179
11.2 Remove View Signal from Disk . . . 180
11.3 Retrieve Signal Dimension from the Parameter File . . . 180
11.4 Retrieve Signal Dimensionality from the Parameter File . . . 180
11.5 Retrieve Signal Interval from the Parameter File . . . 181
11.6 Determine If A File Is An IDP++ File . . . 181
11.7 Open a View File for Reading . . . 182
11.8 Retrieve Signal Origin from the Parameter File . . . 183
11.9 Read Parameters from .spr File . . . 183
11.10 Read View Data File . . . 184
11.10.1 Read Data As One of IDP++'s Types . . . 184
11.10.2 Read Data without Conversion . . . 185
11.11 Read Rows and Columns of 2D View Files . . . 186
11.11.1 Read a Column . . . 186
11.11.2 Read a Row . . . 187
11.12 Resolve a File Name to a View File Name . . . 188
11.13 Saving . . . 189
11.14 Read Signal Type from a Parameter File . . . 190
11.15 Write Parameters . . . 191

12 Support Functions **193**
12.1 Get/Set Points Within Memory . . . 193
12.2 Maximum/Minimum of Two Numbers . . . 194
12.3 Maximum/Minimum of an Array . . . 194
12.4 Median of an Array . . . 194
12.5 Sort an Array . . . 195
12.6 Uniformly Distributed Random Variable . . . 195
12.7 Gamma Distributed Random Variable . . . 195
12.8 Gaussian Distributed Random Variable . . . 196
12.9 Single Point Spline Interpolation . . . 196

13 Global Constants **197**
13.1 Data Type Constants . . . 197
13.2 Axis Constants . . . 197
13.3 Constants Used in Various Functions . . . 198
13.4 Miscellaneous Constants . . . 199

A What's New Since Version 3.5? **201**

B References **203**

Index **204**

1 Changes and New Functions

1.1 Changes to Existing Functions Since Version 3.5

- Signal declarations no longer require a data type declaration e.g. `SIGNAL<float>`, since they are no longer templates. In fact, the declaration `SIGNAL<type>` will generate a compiler error.
- The signal constructors no longer require a data type argument. These are now hard-wired.
- Mixed-type arithmetic is now supported between all four signal types, all complex types, and all scalar types.
- The **gauss** function has been renamed to **mkgauss** to have a name consistent with the other function generation functions.
- The **typeof** function has been renamed **stypeof** because of a conflict with Gun's g++ compiler which has a `typeof` function.
- Added a new set of FFT functions whose functionality is controlled by the logical OR of a set of predefined constants. I implemented these because I could not remember the long calling sequence of the old functions. An convoluted consequence of this is I had to make all the arguments of the old set of FFT functions mandatory, i.e. there are no default arguments.
- The **normalize** function normalizes a signal to the magnitude of the signal sum or maximum. In addition, it offers three new methods of normalization: 1) Sum of magnitude of the signal elements; 2) maximum of the signal element magnitudes; and, 3) signal mean. The **ipnormalize** function performs an in-place normalization.

1.2 New Functions and Features

- The **flatten** function "flattens" a multi-dimensional signal to one dimension. For example, a two dimensional signal of size N_x by N_y will be returned as a one dimensional signal of size $N_x \times N_y$.
- The **fold** function "folds" a one dimensional signal to a multi-dimensional signal of specified dimensions. So a one dimensional signal with $N_x \times N_y$ elements can be folded to become a two dimensional of size N_x by N_y (or N_y by N_x).
- **mkchirp** generates chirps.
- **mkellipse** creates a filled ellipse in a manner similar to `mkcircle`.
- **mkeuler** creates a complex signal using the Euler exponential, e^{ix}.
- **mkramp** generates ramps.
- **mptori** converts two real numbers representing the magnitude and phase of a complex number to a complex number in real and imaginary format.

  Converts two real signals representing the magnitude and phase of a complex signal to a complex signal in real and imaginary format.
- **ndifferentiate** provides a higher order approximation to the first derivative than the `differentiate` function.
- **pdf, pdfh, pdfi** provide different ways of estimating a probability density function from a data set.
- **replicate** creates a two dimensional signal by replicating a one dimensional signal along one of the axes.
- **sumfabs** and **sumint** sum the magnitudes and intensities (magnitude squared) of the signal elements, respectively.

- **snr** computes the signal-to-noise ratio of a data set using a specified signal template.
- There are several types of order statistic filters. See Section 7.41.
- **centerpad** symmetrically zero pads a signal around the edges.
- There are a set of functions used to perform cubic spline interpolation. See Sections 7.43 and 12.9.
- **classArandom** and **Krandom** generate random deviates with distributions following the two-term approximation to Class A and the K-Distribution, respectively.
- The functions **getbase** and **getdir** operate on Unix path strings and return the base name and directory, respectively, of the argument.
- **iptranspose, ipnormalize, ipdemean,** and **ipshsort** functions perform in-place signal transpose, normalization, demeaning, and shell sorting, respectively. The in-place functions over-write the original data and can be used to conserve memory.
- Added a suite of matrix operations. See Section 7.45.

2 Introduction

IDP++ is a multi-dimensional, four-data type signal processing package. Once built and installed it consists of a principal function library, `idp.a`, and a header file containing the package's declarations, `idp.h`. IDP++ consists of four major classes:

`String`	A character string class. The String class offers a set of functions to make the manipulation of strings easier. It eliminates the burdensome need to allocate memory and fill it with a lexical string with functions such as `strcat` and `sprintf`. To use only the `String` class, include `String.h` in your sources and link with `libm.a`. See Section 10 for the details on the String class.
`Array`	A templated (parameterized) array class. To use only the Array class, include Array.h in your sources and link with `libm.a`. See Section 8 for the details on the Array class.
`Complex`	The complex classes which consist of `complex`, a floating point complex type, `dcomplex`, a double complex type, and `Complex<`*type*`>`, a templated (parameterized) complex class. To use only the complex classes, include `Complex.h` in your sources and link with `libm.a`. See Section 9 for the details on the Complex class.
Signal	The signal classes which consist of the four supported data types: `fsignal`, `dsignal`, `csignal`, `dcsignal`. To use any of the signal classes, include `Signal.h` in your sources and link with `viewio.o` and `Signal.o`. This includes the `String`, `Array`, and `Complex` classes. The signal classes support up to four dimensions, nominally X, Y, Z, and t (although the axes can represent any type of units).

The definitions of the four signal data types are:

`fsignal`	A single precision floating point signal whose primitive type is `float`;
`dsignal`	A double precision floating point signal whose primitive type is `double`;
`csignal`	A single precision floating point complex signal whose primitive type is `complex`, that is two consecutive `floats`;
`dcsignal`	A double precision floating point complex signal whose primitive type is `dcomplex`, that is two consecutive `doubles`.

Internally, IDP++ signals must belong to one of the four classes listed above. However IDP++ can read data stored on disk any of the ten following data types:

`char`	An 8-bit data type.
`short`	Nominally a 16-bit (2 byte) data type but may depend upon the computer system. On SUNs and Silicon Graphics machines it is 2 bytes.
`long`	A long data type. On SUNs and Silicon Graphics machines it is 4 bytes.
`float`	A 4-byte floating point data type.
`complex`	An 8-byte floating point data type with real and imaginary parts. This is defined in the Complex class.
`int`	A 4-byte signed integer data type.
`double`	An 8-byte floating point data type.
`unsigned char`	An unsigned 8-bit data type.
`dcomplex`	A 16-byte floating point data type with real and imaginary parts. This is defined in the Complex class
`uccomplex`	An unsigned 16-bit complex data type. This is defined in the Complex class.

3 Installing IDP++

Presumably you received the IDP++ sources in a tar file. Un-tarring it will create the IDP++ version 4.1 directory, `idp++/v4.1`. Within that directory you will find:

`doc`	The documents directory containing the LaTeX sources for this document.
`fortran`	The directory of FORTRAN sources.
`scripts`	The shell scripts directory all of which can be ignored except for `Mktree` which you will have to run before building the IDP++ archive.
`sgi`	The directory for SGI objects and archive.
`src`	The directory of the C++ sources.
`sun`	The directory for Sun objects and archive.
`test`	The directory of test and stand-alone codes.

These are the steps for building the archive for either the Sun or SGI:

1. Run the `Mktree` script from within the `v4.1` directory. This sets the paths within the various `Makefiles` to point to the current directory, and creates required sub-directories and symbolic links;
2. Depending upon your system, `cd` to either `sun` or `sgi` and type "`make -f Makefile.`*mach*" where *mach* is either "`sun`" or "`sgi`." This will build the `idp.a` archive.

4 Writing, Compiling, and Linking

Write IDP++ code as you would ANSI C or C++ but compile and link with `CC` and not `cc`.
To use the `Complex`, `String`, or `Array` classes, either alone or together, include the appropriate header file (`Complex.h`, `String.h`, or `Array.h`) in each of the sources and link with `libm.a`. These classes are stand-alone and do not depend upon `idp.a`.
There is no need to include `stdio.h`, `iostream.h`, or `math.h`. They are included within `Complex.h`, `String.h`, and `Array.h`.
To use the signal classes, include `Signal.h` within each source file and link with `Signal.o`, `viewio.o`, and `libm.a`.
To use the full IDP++ function set, include `idp.h` within each source file and link with `idp.a`, `libm.a`, `libM77.a`, and `libF77.a`.
The `src` directory contains `Makefile.sample` which shows how to create a target, `foo`. Follow the directions therein.

5 Notation Conventions

Throughout this manual the following notation conventions are used

Notation	Definition
signal	Represents one of the four signal data types: `fsignal`, `dsignal`, `csignal`, or `dcsignal`. This notation is used where a function or an operator can be applied to all four types.
stype	Represents the primitive data type of a signal or template class. For example, the *stype* of an `fsignal` is `float`.
op	Represents one of the operators, `+`, `-`, `*`, or `/`.
type	Represents any primitive type, e.g. `float`, `int`, `complex`.

6 Signal Class

6.1 Properties

To date, an IDP++ signal has the following properties:

Property	Description
Name	The name given to the signal, usually the name of the disk file containing the data. The name will be set to the specified file name when the signal is saved using the `save` function. It can be accessed using the `nameof` function.
Directory	The directory path to the signal within the file system. The directory will be set to the specified directory path when the signal is saved using the `save` function. It can be accessed using the `dirof` function.
Data type	This is an `int` representing one of the four supported data types. These, along with the other recognized data types, are listed in the subsection on Global Constants, Section 13. The type property can be accesses using the `stypeof` function.
Data pointer	Pointer to the data within memory. The data pointer can be accessed either via a cast to a pointer of one of the four primitive types (`float`, `double`, `complex`, or `dcomplex`) or via the `ptrto` function.
Dimensionality	The dimensionality of the data. IDP++ recognizes data sets of up to four dimensions. This must be set for a signal to be valid. The signal dimensionality can be accessed via the `ndimof` function.
Dimensions	Size of each of the dimensions. No memory can be allocated for data until the dimensions are set. The signal dimensions can be accessed using the `dimof` function.
Origin	Origin of each of the dimensions. If not set, the origins of all dimensions default to 0. The signal origins can be accessed using the `originof` function.
Interval	Interval of each of the dimensions. If not set, the intervals of all dimensions default to 1. The signal intervals can be accessed using the `intervalof` function.
Time	Time origin of the data in UNIX time format.
Size	Total number of data points or elements within the signal. This is the product of the dimension sizes. The signal size can be accessed with the `nelemof` function.
Comments	Any comments associated with the signal.
File pointer	C stream pointer to an open file containing the signal data. This is used in incremental reads and writes when the data file exceeds memory limitations.
Seek size	Amount to seek through a data file for signals incrementally read from disk.
Seek reference	Seek reference indication from where the data file seek should be performed. Please refer to the UNIX `man` page on `(3S) fseek`.

In most cases, the majority of these properties can be ignored. As a minimum, however, for a signal to be valid, the dimensionality and dimensions must be set. Unless set, no memory is allocated for data.

6.2 Constructors

6.2.1 Default Constructor

signal	(void)

DESCRIPTION:
General constructor. None of the properties are set except the data type. The other properties may be set by any of the following:

- The signal appears on the left hand side of an equation. See Section 6.3 on assignment operators;
- The properties are explicitly set using the member access functions described in Section 6.6;
- The signal is used as an argument in a function which will set or return a value in it.

EXAMPLE:
This code sequence shows default constructor usage. The output of the show function shows none of the properties have been set except for the signal type.

```
fsignal  x ;
dsignal  d ;
csignal  c ;
dcsignal z ;

show( "fsignal from default constructor" , x ) ; fprintf(stdout,"\n") ;
show( "dsignal from default constructor" , d ) ; fprintf(stdout,"\n") ;
show( "csignal from default constructor" , c ) ; fprintf(stdout,"\n") ;
show( "dcsignal from default constructor" , z ) ;
```

The output from the show function:

```
fsignal from default constructor = {
   Name     = { 0 , 0x0 }
   Dir      = { 0 , 0x0 }
   Data     = 0x0
   Type     = 3 float
   NDim     = 0
   Dim      = 0xaf6e8 { }
   Origin   = 0xaf700 { }
   Interval = 0xaf718 { }
   Size     = 0
}

dsignal from default constructor = {
   Name     = { 0 , 0x0 }
   Dir      = { 0 , 0x0 }
   Data     = 0x0
   Type     = 6 double
   NDim     = 0
   Dim      = 0xaf730 { }
   Origin   = 0xaf748 { }
   Interval = 0xaf760 { }
   Size     = 0
```

```
}

csignal from default constructor = {
   Name     = { 0 , 0x0 }
   Dir      = { 0 , 0x0 }
   Data     = 0x0
   Type     = 4 complex
   NDim     = 0
   Dim      = 0xaf778 { }
   Origin   = 0xaf790 { }
   Interval = 0xaf7a8 { }
   Size     = 0
}

dcsignal from default constructor = {
   Name     = { 0 , 0x0 }
   Dir      = { 0 , 0x0 }
   Data     = 0x0
   Type     = 8 dcomplex
   NDim     = 0
   Dim      = 0xaf7c0 { }
   Origin   = 0xaf7d8 { }
   Interval = 0xaf7f0 { }
   Size     = 0
}
```

A signal array can be declared as would any other type of array:

```
fsignal x[ 256 ];    // An array of 256 fsignals.
csignal z[ 100 ];    // An array of 100 csignals.
```

For signal arrays, note the precedence of `[]`:

```
x[ 10 ][ 1 ] = 2.3 ; // Set the second element of the 11th signal.
```

The first set of brackets selects a particular signal in the array, the second set selects an element of the signal.

6.2.2 Copy Constructor

```
fsignal          (const fsignal &Input)
fsignal          (const dsignal &Input)
fsignal          (const csignal &Input)
fsignal          (const dcsignal &Input)

dsignal          (const fsignal &Input)
dsignal          (const dsignal &Input)
dsignal          (const csignal &Input)
dsignal          (const dcsignal &Input)

csignal          (const fsignal &Input)
csignal          (const fsignal &Real, const fsignal &Imag)
csignal          (const dsignal &Input)
csignal          (const dsignal &Real, const dsignal &Imag)
csignal          (const csignal &Input)
csignal          (const dcsignal &Input)

dcsignal         (const fsignal &Input)
dcsignal         (const fsignal &Real, const fsignal &Imag)
dcsignal         (const dsignal &Input)
dcsignal         (const dsignal &Real, const dsignal &Imag)
dcsignal         (const csignal &Input)
dcsignal         (const dcsignal &Input)
```

DESCRIPTION:
Copy constructor. Creates a signal and initializes it to the argument. Both member information and data are copied to the new signal. As indicated, you can initialize any signal type by any other type. The following rules are used:

- When initializing from a complex type to a real type, only the real part of the complex signal is preserved;
- When initializing from a real type to a complex type, the real part is set to the real part of the signal and the imaginary part is set to zero;
- When initializing from two real types to a complex type; the real part is set to the first real signal, and the imaginary part is set to the second real signal.

EXAMPLE:

```
fsignal x, y ;
dsignal d( x ) ;
csignal Z( x , y ) ;
dcsignal DZ( y ) ;
```

6.2.3 Detailed Constructor

```
signal          (const int NDim ,
                 const int *Dim ,
                 const stype *Data = 0,
                 const float *Origin = 0,
                 const float *Interval = 0)
```

DESCRIPTION:
General signal constructor. This flexible constructor may be used to create a signal with any of its major parameters set. At a minimum the dimensionality and dimensions must be specified in the `NDim` and `Dim` arguments respectively. Setting only these two parameters creates the signal and allocates memory.
The arguments are:

`NDim`	The dimensionality of the signal. Must be within the range of 1 to `MAXDIM`, where `MAXDIM` is four. See Section 13 on Global Constants.
`Dim`	An array of at least size `NDim` specifying the sizes of each of the dimensions. With the `NDim` and `Dim` arguments, the constructor allocates data memory and sets the number of signal elements to the product `Dim[0]*···*Dim[NDim-1]` If the `Data` argument is not set, the memory is initialized to zero.
`Data`	A pointer to an area of memory of at least size `Dim[0]*···*Dim[NDim-1]*sizeof(`*stype*`)` containing the data to be copied to the signal, where *stype* is one of `float`, `double`, `complex`, or `dcomplex` depending upon the signal data type.
`Origin`	An array of at least size `NDim` specifying the origins of each of the signal's axes. If not specified or set to zero, the origins default to zero.
`Interval`	An array of at least size `NDim` specifying the intervals of each of the signal's axes. If not specified or set to zero, the intervals default to one.

EXAMPLE:
This code sequence shows how signals can be created, memory allocated, and properties set

```
int
  xdim    = 100 ,
  ddim[2] = { 10 , 12 } ,
  cdim[3] = { 10 , 12 , 14 } ,
  zdim[4] = { 10 , 12 , 14 , 16 } ;
float
  origin[4]   = { -1 , 0 , Pi , 10 },
  interval[4] = { 0.5 , 1 , Pi/10. , 1./3. } ;

fsignal  x( 1, &xdim ) ;
dsignal  d( 2,  ddim, 0, origin ) ;
csignal  c( 3,  cdim, 0, origin, interval ) ;
dcsignal z( 4,  zdim, 0, origin, interval ) ;

show( "fsignal from detailed constructor" , x ) ; fprintf(stdout,"\n") ;
```

```
    show( "dsignal from detailed constructor" , d ) ; fprintf(stdout,"\n") ;
    show( "csignal from detailed constructor" , c ) ; fprintf(stdout,"\n") ;
    show( "dcsignal from detailed constructor" , z ) ;
```

The output of the show function is

```
fsignal from detailed constructor = {
   Name     = { 0 , 0x0 }
   Dir      = { 0 , 0x0 }
   Data     = 0xafce8
   Type     = 3 float
   NDim     = 1
   Dim      = 0xaf6e8 { 100 }
   Origin   = 0xaf7f0 { 0 }
   Interval = 0xaf7d8 { 1 }
   Size     = 100
   MIN/MAX  = 0/0
}

dsignal from detailed constructor = {
   Name     = { 0 , 0x0 }
   Dir      = { 0 , 0x0 }
   Data     = 0xafe80
   Type     = 6 double
   NDim     = 2
   Dim      = 0xaf7c0 { 10 12 }
   Origin   = 0xaf7a8 { -1 0 }
   Interval = 0xaf790 { 1 1 }
   Size     = 120
   MIN/MAX  = 0/0
}

csignal from detailed constructor = {
   Name     = { 0 , 0x0 }
   Dir      = { 0 , 0x0 }
   Data     = 0xb0248
   Type     = 4 complex
   NDim     = 3
   Dim      = 0xaf778 { 10 12 14 }
   Origin   = 0xaf760 { -1 0 3.14159 }
   Interval = 0xaf748 { 0.5 1 0.314159 }
   Size     = 1680
   MIN/MAX  = 0/0
}

dcsignal from detailed constructor = {
   Name     = { 0 , 0x0 }
   Dir      = { 0 , 0x0 }
   Data     = 0xb38d8
   Type     = 8 dcomplex
   NDim     = 4
   Dim      = 0xaf730 { 10 12 14 16 }
   Origin   = 0xaf718 { -1 0 3.14159 10 }
   Interval = 0xaf700 { 0.5 1 0.314159 0.333333 }
```

```
   Size      = 26880
   MIN/MAX   = 0/0
}
```

Once allocated, signal memory can be used for subsequent processing:

```
     int dim = 100 ;
     double *ptr ;
     dsignal ramp(1,&dim) ;
// Cast the signal to a pointer which retrieves the pointer to the data.
     ptr = ramp ;
     for( int n=0 ; n<dim ; n++ )
          *ptr++ = -4 + 3*n ;
```

A signal array with set parameters can be defined using the cast form of the constructor:

```
     fsignal x[ 256 ];     // An array of 256 fsignals.

     for( int n=0 ; n<256 ; n++ )
          x[ n ] = fsignal(ndim,dim,0,origin,interval) ;
```

6.2.4 Size Constructor

signal	(const int NDim, const int Nx, const int Ny = 0, const int Nz = 0, const int Nt = 0)

DESCRIPTION:
General constructor using a list of scalar values instead of a pointer to specify the dimensions. Memory is allocated and initialized to zero. Use this when it is too bothersome to declare an `int` array. The origins and intervals are set to their default values of 0 and 1 respectively.
The arguments are:

`NDim`	The dimensionality of the signal. Must be within the range of 1 to `MAXDIM`, where `MAXDIM` is four. See Section 13 on Global Constants.
`Nx`	The size of the X-dimension.
`Ny`	The size of the Y-dimension.
`Nz`	The size of the Z-dimension.
`Nt`	The size of the T-dimension.

EXAMPLE:
This code sequence shows size constructor usage. The output of the `show` function show the signal properties. The origins and intervals have been set to their default values of 0 and 1, respectively.

```
fsignal  x(1,100) ;
dsignal  d(2,10,20) ;
csignal  c(3,64,128,150) ;
dcsignal z(4,30,15,10,20) ;

show( "fsignal from size constructor" , x ) ; fprintf(stdout,"\n") ;
show( "dsignal from size constructor" , d ) ; fprintf(stdout,"\n") ;
show( "csignal from size constructor" , c ) ; fprintf(stdout,"\n") ;
show( "dcsignal from size constructor" , z ) ;
```

The output from the `show` function:

```
fsignal from size constructor = {
   Name     = { 0 , 0x0 }
   Dir      = { 0 , 0x0 }
   Data     = 0xafce8
   Type     = 3 float
   NDim     = 1
   Dim      = 0xaf6e8 { 100 }
   Origin   = 0xaf7f0 { 0 }
   Interval = 0xaf7d8 { 1 }
   Size     = 100
   MIN/MAX  = 0/0
}

dsignal from size constructor = {
```

```
   Name     = { 0 , 0x0 }
   Dir      = { 0 , 0x0 }
   Data     = 0xafe80
   Type     = 6 double
   NDim     = 2
   Dim      = 0xaf7c0 { 10 20 }
   Origin   = 0xaf7a8 { 0 0 }
   Interval = 0xaf790 { 1 1 }
   Size     = 200
   MIN/MAX  = 0/0
}

csignal from size constructor = {
   Name     = { 0 , 0x0 }
   Dir      = { 0 , 0x0 }
   Data     = 0xb04c8
   Type     = 4 complex
   NDim     = 3
   Dim      = 0xaf778 { 64 128 150 }
   Origin   = 0xaf760 { 0 0 0 }
   Interval = 0xaf748 { 1 1 1 }
   Size     = 1228800
   MIN/MAX  = 0/0
}

dcsignal from size constructor = {
   Name     = { 0 , 0x0 }
   Dir      = { 0 , 0x0 }
   Data     = 0xa106d8
   Type     = 8 dcomplex
   NDim     = 4
   Dim      = 0xaf730 { 30 15 10 20 }
   Origin   = 0xaf718 { 0 0 0 0 }
   Interval = 0xaf700 { 1 1 1 1 }
   Size     = 90000
   MIN/MAX  = 0/0
}
```

A signal array with allocated memory can be defined using the cast form of the constructor:

```
fsignal x[ 256 ];    // An array of 256 fsignals.

for( int n=0 ; n<256 ; n++ )
     x[ n ] = fsignal(2, 100, 128)   // An array of 100x128 fsignals.
```

6.2.5 File Constructor

signal	(const char *DiskFileName, const int *Start = (int*)0, const int *Size = (int*)0)

DESCRIPTION:
Constructor using the information from an existing View disk file. The character string is the file name. This constructor creates a signal from the parameters and data stored in the `.spr` and `.sdt` disk files.
`Start` and `Size` are arrays specifying the starting location and size for partial reads.
Any of the ten data types described in the introduction can be read in and cast to the particular signal data type.
The following rules are used when casting between types:

- When casting from a complex type to a real type, only the real part of the complex signal is preserved;
- When casting from a real type to a complex type, the real part is set to the real part of the signal and the imaginary part is set to zero.

EXAMPLE:

```
csignal R( "radar") ;
dsignal sin( "sinewave" ) ;

int
   location[2] = { 100, 200 } ,
   size[2]     = { 1000, 1000 } ;

fsignal chunk( "radar" , location , size) ;
```

A signal array defined from data files can be created using the cast form of the constructor:

```
fsignal rgb[ 3 ];    // An array of 3 fsignals.

rgb[ 0 ] = fsignal( "red" ) ;
rgb[ 1 ] = fsignal( "green" ) ;
rgb[ 2 ] = fsignal( "blue" ) ;
```

Alternatively, you may use the cast from a string:

```
rgb[ 0 ] = "red"  ;
rgb[ 1 ] = "green"  ;
rgb[ 2 ] = "blue"  ;
```

6.2.6 Mixed Constructor

signal	(const *stype* *Data, const *signal* &Input)

DESCRIPTION:
Constructor using parameters from an existing signal. This constructor creates a signal using the parameters from the existing signal Input and the data pointed to by Data. If the total number of data points in Input is *Size*, then Data must point to at least *Size**sizeof(*stype*) bytes, where *stype* is the particular data type.

6.3 Assignment Operators

6.3.1 Signal to Signal Assignment Operators

```
fsignal&        operator =        (const fsignal &Input)

dsignal&        operator =        (const fsignal &Input)
dsignal&        operator =        (const dsignal &Input)

csignal&        operator =        (const fsignal &Input)
csignal&        operator =        (const dsignal &Input)
csignal&        operator =        (const csignal &Input)

dcsignal&       operator =        (const fsignal &Input)
dcsignal&       operator =        (const dsignal &Input)
dcsignal&       operator =        (const csignal &Input)
dcsignal&       operator =        (const dcsignal &Input)
```

DESCRIPTION:
Signal to signal assignment operator. All parameters and data are copied to the destination signal.
These operators ensure casting from a lower data type to a higher data type is performed implicitly. Casting from a higher data type to a lower data type must be performed explicitly by using the constructors from Section 6.2.2

EXAMPLE:
In its most simple form, the assignment operator is used to create a copy of a signal:

```
csignal A , B ;

// Set A somewhere...

B = A ;
```

It can also be used when creating new signals. As the result of a function call, for example:

```
fsignal demean(const fsignal &Input)
{
    fsignal Result ;

    Result = demean( Input ) ;

    return Result ;
 }
```

Or when memory is needed:

```
dsignal d ;

d = dsignal(3, 100, 100, 128) ;
```

6.3.2 File to Signal Assignment Operators

signal&	operator =	(const char *DiskFileName)

DESCRIPTION:
Character string to signal assignment operator. The destination signal has its parameters and data set using the parameters and data of the disk file name pointed to by `DiskFileName`.
This performs a full read of the data. For a partial read see the file constructor of Section 6.2.5.
signal is one of the four signal types: `fsignal`, `dsignal`, `csignal`, `dcsignal`.

EXAMPLE:

```
fsignal radar ;

/*
 * Read entire data file
 */
radar = "HH_tai" ;

int
    start[2] = { 0, 2000 },
    size[2]  = { 1024, 10000 } ;

/*
 * Read part of data file using the file constructor as a cast
 */
 radar = fsignal( "HH_tai" , start , size ) ;
```

6.3.3 Scalar to Signal Assignment Operator

signal&	operator =	(const *stype* Scalar)

DESCRIPTION:
Constant to signal assignment. If the destination signal has been previously set, each data element of it will be assigned the value `Scalar`. The previous data will be lost. The dimensionality, dimension sizes, origins, intervals, etc. remain unchanged. If the destination signal has not been set, it will become a one dimensional, single element signal whose value is `Scalar`.

6.4 Assignment Arithmetic Operators

6.4.1 Signal to Signal Assignment Arithmetic Operators

```
fsignal&        operator op=      (const fsignal &Input)
fsignal&        operator op=      (const dsignal &Input)

dsignal&        operator op=      (const dsignal &Input)
dsignal&        operator op=      (const fsignal &Input)

csignal&        operator op=      (const fsignal &Input)
csignal&        operator op=      (const dsignal &Input)
csignal&        operator op=      (const dcsignal &Input)

dcsignal&       operator op=      (const fsignal &Input)
dcsignal&       operator op=      (const dsignal &Input)
dcsignal&       operator op=      (const csignal &Input)
```

DESCRIPTION:
Assignment arithmetic operators from one signal to another where *op* is one of +, -, *, /. The operators are applied on an element-by-element basis.

6.4.2 Scalar to Signal Assignment Arithmetic Operators

```
signal&         operator op=      (const stype Scalar)
```

DESCRIPTION:
Assignment arithmetic operators from a scalar to a signal where *op* is one of +, -, *, /. The arithmetic operator is applied to each element of the signal.

6.5 Binary Operators

6.5.1 Signal-Signal Binary Operators

fsignal	operator *op*	(const fsignal &A, const fsignal &B)
dsignal	operator *op*	(const dsignal &A, const dsignal &B)
dsignal	operator *op*	(const dsignal &A, const fsignal &B)
dsignal	operator *op*	(const fsignal &A, const dsignal &B)
csignal	operator *op*	(const csignal &A, const fsignal &B)
csignal	operator *op*	(const fsignal &A, const csignal &B)
csignal	operator *op*	(const csignal &A, const dsignal &B)
csignal	operator *op*	(const dsignal &A, const csignal &B)
dcsignal	operator *op*	(const dcsignal &A, const dcsignal &B)
dcsignal	operator *op*	(const dcsignal &A, const fsignal &B)
dcsignal	operator *op*	(const fsignal &A, const dcsignal &B)
dcsignal	operator *op*	(const dcsignal &A, const dsignal &B)
dcsignal	operator *op*	(const dsignal &A, const dcsignal &B)
dcsignal	operator *op*	(const dcsignal &A, const csignal &B)
dcsignal	operator *op*	(const csignal &A, const dcsignal &B)

DESCRIPTION:
Binary arithmetic operators between signals where *op* is one of +, -, *, /. The operators are applied on an element-by-element basis.

6.5.2 Signal-Scalar Binary Operators

signal	operator *op*	(const *signal* &A, const *stype* Scalar)
signal	operator *op*	(const *stype* Scalar, const *signal* &A)

DESCRIPTION:
Binary arithmetic operators between signals and scalars where *op* is one of +, -, *, /.

6.6 Miscellaneous Signal Functions and Operators

6.6.1 Signal Element Access

stype&	*signal* [] (int n)

DESCRIPTION:
Signal element access. The operator can be used to set and retrieve signal elements. For 2 or higher dimensional signals, the address calculation must be performed manually. Range checking is performed.

EXAMPLE:

```
fsignal A( "Temperature" ) ;
int x ;
A[ 2 ] = 3 ;        // Set an element
x      = A[ 2 ] ;   // Retrieve an element
```

6.6.2 Signal Casts

*stype**	(*signal*)

DESCRIPTION:
Cast to pointer, returns the pointer to the signal data. Note, this cast may not work on signals declared as `const`. In those cases, use the `ptrto()` function of Section 6.6.9 to retrieve the pointer to the signal data.

EXAMPLE:

```
csignal Z( "HH_c" ) ;
complex *zptr ;
zptr = Z ;  // zptr now points to the data of Z
```

If the compiler is efficient at understanding the difference between non-`const` and `const` types, then you will be able to use the following for signals declared as `const`:

```
void foo(const csignal &Z)
{
     complex const *zptr = Z ;

     // . . .
```

6.6.3 Signal Unary Negation

signal	operator -	(const *signal* &A)

DESCRIPTION:
Unary negation. All data elements of the signal are negated.

6.6.4 Number of Elements

int&	nelemof	(*signal* &A)

DESCRIPTION:
Function to retrieve or set the number of elements in a signal. If an `ndim`-dimensional signal has dimensions `dim[0]`, `dim[1]`, ... `dim[ndim-1]`, `nelemof` will return `dim[0]*dim[1]*...*dim[ndim-1]`.

6.6.5 Signal Dimensionality Access

int&	ndimof	(*signal* &A)

DESCRIPTION:
Function to retrieve or set the dimensionality of a signal. The dimensionality, `NDim`, must be $1 \leq$ `NDim` $\leq$ `MAXDIM`.

6.6.6 Signal Dimension Size Access

int&	dimof	(*signal* &A, const int Axis)
Array<int>&	dimof	(*signal* &A)

DESCRIPTION:
Functions to retrieve or set the dimensions of a signal. The first form retrieves or sets the dimension specified by axis. The second form returns an Array<int> of size MAXDIM containing all dimension sizes. Axis is one of the axis specifiers described in the subsection on Global Constants, Section 13, X_AXIS, Y_AXIS, Z_AXIS, or T_AXIS.

EXAMPLE:

```
fsignal rar( "HH_tai") ;
Array<int> dim ;
int Nx ;
dim = dimof( rar ) ;
Nx  = dimof( rar , X_AXIS ) ;
```

6.6.7 Signal Origin Access

float&	originof	(*signal* &A, const int Axis)
Array<float>&	originof	(*signal* &A)

DESCRIPTION:
Functions to retrieve or set the origins of a signal. The first form retrieves or sets the origin specified by Axis. The second form returns an Array<float> of size MAXDIM containing all origins. Axis is one of the axis specifiers described in the subsection on Global Constants, Section 13, X_AXIS, Y_AXIS, Z_AXIS, or T_AXIS.

EXAMPLE:

```
fsignal rar( "HH_tai") ;
Array<float> origin ;
float x0
origin = originof( rar ) ;
x0     = originof( rar , X_AXIS ) ;
```

Note: This function can also be used as an lvalue:

```
csignal Z(2, 100, 100) ;
originof(Z, X_AXIS) = -Pi ;
originof(Z, Y_AXIS) = -Pi ;
```

6.6.8 Signal Interval Access

`float&`	`intervalof`	(*signal* `&A, const int Axis`)
`Array<float>&`	`intervalof`	(*signal* `&A`)

DESCRIPTION:
Functions to retrieve or set the intervals of a signal. The first form retrieves or sets the interval specified by **Axis**. The second form returns an **Array<float>** of size **MAXDIM** containing all intervals. **Axis** is one of the axis specifiers described in the subsection on Global Constants, Section 13, **X_AXIS**, **Y_AXIS**, **Z_AXIS**, or **T_AXIS**.

EXAMPLE:

```
fsignal rar( "HH_tai") ;
Array<float> interval ;
float dx
interval = intervalof( rar ) ;
dx       = intervalof( rar , X_AXIS ) ;
```

Note: This function can also be used as an lvalue:

```
csignal Z(2, 100, 100) ;
intervalof(Z, X_AXIS) = 0.1 ;
intervalof(Z, Y_AXIS) = 1.5 ;
```

6.6.9 Signal Data Member Access

stype *	ptrto	(const *signal* &A, const int n = 0)

DESCRIPTION:
Function to retrieve the pointer to the data of a signal. Range checking is performed. n specifies an offset within the data, if not specified the pointer to the first element is returned. You may need to use ptrto() over the cast to pointer of Section 6.6.2 when working with signals declared as const.

EXAMPLE:
This will always work:

```
const fsignal R( "rar") ;
register float *rptr ;
rptr = ptrto( R ) ;

for( int n=0 ; n<nelemof(R) ; n++ )
   fprintf(stdout, "%g\n", *rptr++) ;
```

Some compilers may complain about this

```
const fsignal R( "rar") ;
register float const *rptr = R ;

for( int n=0 ; n<nelemof(R) ; n++ )
   fprintf(stdout, "%g\n", *rptr++) ;
```

6.6.10 Signal Name Member Access

String	nameof	(const *signal* &A)
void	nameof	(*signal* &A, const char *Name)

DESCRIPTION:
Functions to retrieve or set the name of a signal. The first form retrieves the signal name. The second form is used to set it.
Note: When using this function in a printf, fprintf, etc, function to write a signal's name, you must cast the first form to a char*:

```
fsignal x ;
nameof(x, "Nobody") ;

fprintf(stdout, "My name is %s\n", (char*)nameof(x)) ;
```

6.6.11 Signal Directory Member Access

String	dirof	(const *signal* &A)
void	dirof	(*signal* &A, const char *Directory)

DESCRIPTION:
Functions to retrieve or set the directory path of a signal. The first form retrieves the signal directory. The second form is used to set it.
Note: When using this function in a `printf`, `fprintf`, etc, function to write a signal's name, you must cast the first form to a `char*`:

```
fsignal x = "/usr/people/lehman/data/test1" ;

fprintf(stdout, "I live in %s\n", (char*)dirof(x)) ;
```

6.6.12 Signal Type Member Access

int&	stypeof	(*signal* &A)

DESCRIPTION:
Function to retrieve or set the data type of a signal. This returns one of the four signal types: `FLOAT`, `DOUBLE`, `COMPLEX`, `DCOMPLEX`. The signal data types are described in the subsection on Global Constants, Section 13.
Note: This function was formerly called `typeof` but the name was changed to avoid a conflict with a GNU C++ (gcc) built-in of the same name.

6.6.13 Signal Type Member Access

`const char*`	`gettype`	`(const` *signal* `&A)`

DESCRIPTION:
Function to retrieve a `const char` pointer to the name of the signal data type. This returns a pointer to a character string to one of the signal type names described in the subsection on Global Constants, Section 13: "float", "double", "complex", or "dcomplex".

6.6.14 Signal Minimum

stype	`minimum`	`(const` *signal* `&A)`
stype	`minimum`	`(const` *signal* `&A, int *Location)`

DESCRIPTION:
Return the signal minimum.
The first form returns the value of the signal minimum. The second form returns both the signal minimum and its location. `Location` is an array of at least size `ndimof(A)`.

6.6.15 Signal Maximum

stype	`maximum`	`(const` *signal* `&A)`
stype	`maximum`	`(const` *signal* `&A, int *Location)`

DESCRIPTION:
Return the signal maximum.
The first form returns the value of the signal maximum. The second form returns both the signal maximum and its location. `Location` is an array of at least size `ndimof(A)`.

6.6.16 Signal Sum

stype	sum	(const *signal* &A)

DESCRIPTION:
Return the sum of the elements in a signal.

6.6.17 Signal Constant Fill

stype	constant	(const *signal* &A, const *stype* Constant)

DESCRIPTION:
Function to fill the elements a signal with a constant. The value of the constant is returned.

EXAMPLE:

```
dsignal X(1, 128) ;
csignal Z(2, 128, 200);

constant( X , Pi ) ;

constant( Z , complex(1, 1) ) ;
```

6.6.18 Saving Signals

int	save	(*signal* &A)
int	save	(*signal* &A, const char *DiskFileName)

DESCRIPTION:
Functions to save a signal. The first form uses the signal name for the data file name. The second form uses `DiskFileName`.
The function returns `SUCCESS` if the signal is written correctly, `FAILURE` otherwise. `SUCCESS` and `FAILURE` are described in the subsection on Global Constants, Section 13.
The signal name and directory are modified to reflect the base name and path in `DiskFileName`.
Note: Since the `String` to `char*` is defined you may use expressions such as:

```
String Name( "foobar" ) ;
save( X , Name  ) ;
```

6.6.19 Appending Comments to SPR File

int	AppendToSPRFile	(const *signal* &A, const char *Comments)

DESCRIPTION:
Append comments to the signal's parameter (`.spr`) file.

EXAMPLE:
This code sequence shows how to save the command line arguments to an existing `.spr` file:

```
main(int argn,char *args[])
{
    /*
     * Save command line.
     */
    String Comments = args[0] ;
    for( int n=1 ; n<argn ; n++ )
  Comments += String(" ") + String(args[n]) ;

    dsignal X ;

    // . . .

    /*
     * Note: Saving X as "solution" sets its name parameter.
     */
    save( X , "solution" ) ;

    AppendToSPRFile( X , Comments ) ;
```

6.6.20 Deleting a Signal

`int`	`remove`	`(const` *signal* `&A)`

DESCRIPTION:
Delete the signal from disk.

6.6.21 Listing Signals in ASCII

```
void        ascii        (const signal &Input)
void        ascii        (const signal &Input,
                          const char *Filename,
                          const int Coordinate = WRA_NONE)
void        ascii        (const signal &Input,
                          FILE *Destination,
                          const int Coordinate = WRA_NONE)
```

DESCRIPTION:
Function to display or write a signal in ASCII format. The number format for `floats` and `doubles` is `%16.8e` so that they may be read by FORTRAN. The first form writes the signal to `stdout`. The second writes to the file whose name is specified in `Filename`. The third form writes to the open file pointer `Destination`. The `Coordinate` option specifies whether the signal element coordinates or indices are written with the data. `WRA_NONE` selects no coordinates are to be written, `WRA_COOR` selects the writing of the signal coordinates, `WRA_INDEX` selects the signal element index. These constants are described in the subsection on Global Constants, Section 13.
The arguments are:

`Input`	Input signal.
`Filename`	Name of file where ASCII data should be written.
`Destination`	Open file pointer to where the ASCII data should be written.
`Coordinate`	Optional argument specifying whether the signal coordinates, indices, or neither should be written with the data. Must be one of `WRA_NONE`, `WRA_COOR`, or `WRA_INDEX`. The default is `WRA_INDEX`.

EXAMPLE:

This code sequence demonstrates the three methods of writing signals in ASCII.

```
fsignal X(1, 10) ;
float
     x0 = 0.1,
     dx = 0.2 ;
int n ;

originof( X, X_AXIS )   = x0 ;
intervalof( X, X_AXIS ) = dx ;

for( n=0 ; n<nelemof(X) ; n++ )
     X[n] = pow( x0 + n*dx , 2 ) ;

fprintf(stdout,"X w/out coordinates:\n") ;
ascii( X , stdout , WRA_NONE ) ;

fprintf(stdout,"\nX with coordinates:\n") ;
ascii( X , stdout , WRA_COOR ) ;

fprintf(stdout,"\nX with indices:\n") ;
ascii( X , stdout , WRA_INDEX ) ;
```

The output from this code is

```
X w/out coordinates:
  1.00000007e-02
  9.00000036e-02
  2.50000000e-01
  4.90000069e-01
  8.10000062e-01
  1.21000004e+00
  1.69000018e+00
  2.25000000e+00
  2.89000010e+00
  3.61000037e+00

X with coordinates:
  1.00000001e-01   1.00000007e-02
  3.00000012e-01   9.00000036e-02
  5.00000000e-01   2.50000000e-01
  6.99999988e-01   4.90000069e-01
  8.99999976e-01   8.10000062e-01
  1.10000002e+00   1.21000004e+00
  1.30000007e+00   1.69000018e+00
  1.50000012e+00   2.25000000e+00
  1.70000017e+00   2.89000010e+00
  1.90000021e+00   3.61000037e+00

X with indices:
 0   1.00000007e-02
 1   9.00000036e-02
 2   2.50000000e-01
 3   4.90000069e-01
 4   8.10000062e-01
 5   1.21000004e+00
 6   1.69000018e+00
 7   2.25000000e+00
 8   2.89000010e+00
 9   3.61000037e+00
```

6.6.22 Signal Show

```
void          show          (const char *Message,
                             const signal &Input,
                             FILE *fp = stdout)
```

DESCRIPTION:
Function to show a signal's properties in a fancy manner. Memory addresses are showed as well permitting this function to be used for debugging purposes.

EXAMPLE:
For example, the following code:

```
int
     N = 121,
     n ;
float
     x0 = -Pi ,
     dx = 2*Pi/N ;
fsignal
     c( 1, &N, 0, &x0, &dx ) ,
     s( 1, &N, 0, &x0, &dx ) ;

/*
 * Create a cosine and a sine signal
 */
for( n=0 ; n<N ; n++ )
{
     c[n] = cos( x0 + n*dx ) ;
     s[n] = sin( x0 + n*dx ) ;
}

/*
 * Combine them into a complex signal
 */
csignal
     z(c, s) ;

/*
 * Saving the signals sets their names
 */
save(c, "cosine") ;
save(s, "sine") ;
save(z, "euler") ;

show("Cosine signal", c) ; fprintf(stdout,"\n") ;
show("Sine signal"  , s) ; fprintf(stdout,"\n") ;
show("Euler signal" , z) ; fprintf(stdout,"\n") ;
```

produces

```
Cosine signal = {
   Name     = { 6 , 0x673b8 "cosine" }
```

```
   Dir      = { 0 , 0x0 }
   Data     = 0x669f8
   Type     = 3 float
   NDim     = 1
   Dim      = 0x66518 { 121 }
   Origin   = 0x66530 { -3.14159 }
   Interval = 0x66548 { 0.0519272 }
   Size     = 121
   MIN/MAX  = -1/0.999663
}

Sine signal = {
   Name     = { 4 , 0x673c8 "sine" }
   Dir      = { 0 , 0x0 }
   Data     = 0x66be8
   Type     = 3 float
   NDim     = 1
   Dim      = 0x66560 { 121 }
   Origin   = 0x66578 { -3.14159 }
   Interval = 0x66590 { 0.0519272 }
   Size     = 121
   MIN/MAX  = -0.999916/0.999916
}

Euler signal = {
   Name     = { 5 , 0x673d8 "euler" }
   Dir      = { 0 , 0x0 }
   Data     = 0x66dd8
   Type     = 4 complex
   NDim     = 1
   Dim      = 0x665f0 { 121 }
   Origin   = 0x66608 { -3.14159 }
   Interval = 0x66620 { 0.0519272 }
   Size     = 121
   MIN/MAX  = 1/1
}
```

6.6.23 Sorting Signal Elements

signal	shsort	(const *signal* &Input)
void	ipshsort	(*signal* &Input)

DESCRIPTION:
This functions performs a shell sort on the signal elements from lowest to highest. In the case of complex values, the magnitudes are compared. The returned signal is one dimensional regardless of the input signal's dimensionality. The number of elements is preserved.
`ipshsort` performs an in-place sort in order to save memory.

EXAMPLE:
This code sequence shows how a real and a complex signal are sorted:

```
{
     int
          N = 5,
          n ;
     fsignal x(1, N) ;
     csignal c(1, N) ;
     void list(fsignal&x) ;
     void list(csignal&x) ;

     for( n=0 ; n<N ; n++ )
     {
          x[n] = pow(20 - n,2) ;
          c[n] = complex( sqrt(x[n]) , -sqrt(x[n]) ) ;
     }

     fprintf(stdout,"fsignal before sorting:\n") ;
     list(x) ;
     x = shsort(x) ;
     fprintf(stdout,"fsignal after sorting:\n") ;
     list(x) ;

     fprintf(stdout,"csignal before sorting:\n") ;
     list(c) ;
     c = shsort(c) ;
     fprintf(stdout,"csignal after sorting (by magnitude):\n") ;
     list(c) ;
}
void list(fsignal&x)
{
     for( int n=0 ; n<nelemof(x) ; n++ )
          fprintf(stdout,"     %3.0f\n",x[n]) ;
     fprintf(stdout,"\n") ;
}
void list(csignal&x)
{
     for( int n=0 ; n<nelemof(x) ; n++ )
          fprintf(stdout,"     |(%3.0f,%3.0f)| = %5.2f\n",
                  real(x[n]),imag(x[n]),fabs(x[n])) ;
```

```
    fprintf(stdout,"\n") ;
}
```

The output is

```
fsignal before sorting:
    400
    361
    324
    289
    256

fsignal after sorting:
    256
    289
    324
    361
    400

csignal before sorting:
    |( 20,-20)| = 28.28
    |( 19,-19)| = 26.87
    |( 18,-18)| = 25.46
    |( 17,-17)| = 24.04
    |( 16,-16)| = 22.63

csignal after sorting (by magnitude):
    |( 16,-16)| = 22.63
    |( 17,-17)| = 24.04
    |( 18,-18)| = 25.46
    |( 19,-19)| = 26.87
    |( 20,-20)| = 28.28
```

6.7 Get/Set a Point Within a Signal

These functions are used to get or set an element value within a signal.

stype	getpt	(const *signal* &Input, const int x, const int y = 0, const int z = 0, const int t = 0)
stype&	getpt	(*signal* &Input, const int x, const int y = 0, const int z = 0, const int t = 0)

DESCRIPTION:
The first form is for use on the right hand side of equations while the second form is for the left hand side. Note: Some compilers may object to the overloading of

```
getpt(const fsignal&, ...)
```

and

```
getpt(fsignal, ...)
```

claiming they cannot distinguish between the two. They are wrong. According to the ARM [1] Chapter 13, for any type `T`, it is "... possible to distinguish between `const T&`, `volatile T&`, and plain `T&` so functions that differ only in this respect may be defined."

6.8 Get Rows and Columns of 2D Signals

These functions are used for getting rows and columns of 2 dimensional signals. The first form returns a signal while the second form stores the result in an array.
These functions work only on 2 dimensional signals.

6.8.1 Signal to Signal Get Column

signal	getcol	(const *signal* &Input, const int Column)

DESCRIPTION:
Extract a column from a 2D signal. Column origin and interval are preserved.

6.8.2 Signal to Memory Get Column

void	getcol	(const *signal* &Input, *stype* *Result, const int Column)

DESCRIPTION:
Extract a column from a 2D signal and store the result in the area pointed to by `Result`.

6.8.3 Signal to Signal Get Row

signal	getrow	(const *signal* &Input, const int Row)

DESCRIPTION:
Extract a row from a 2D signal. Row origin and interval are preserved.

6.8.4 Signal to Memory Get Row

```
void        getrow        (const signal &Input,
                           stype *Result,
                           const int Row)
```

DESCRIPTION:
Extract a row from a 2D signal and store the result in the area pointed to by `Result`.

6.9 Store Rows and Columns of 2D Signals

These functions are used for storing a 1 dimensional input signal as rows or columns of a 2 dimensional output signal.

6.9.1 Column Storing

```
void        putcol      (const signal &Input,
                         signal &Out,
                         const int Column)
```

DESCRIPTION:
Insert a 1D signal into a column of a 2D signal, overwriting the data.

6.9.2 Row Storing

```
void        putrow      (const signal &Input,
                         signal &Out,
                         const int Row)
```

DESCRIPTION:
Insert a 1D signal into a row of a 2D signal, overwriting the data.

6.10 Signal Math Functions

These math functions reside solely within the various signal classes and do not require the full IDP++ library (i.e. `idp.a`). If using only these functions it is sufficient to link only with `Signal.o` and `viewio.o`. For example:

```
CC -o foo foo.c /usr/local/idp++/Signal.o /usr/local/idp++/viewio.o -lm
```

6.10.1 Phase of a Complex Signal

`fsignal`	`arg`	`(const csignal &Input)`
`dsignal`	`arg`	`(const dcsignal &Input)`

DESCRIPTION:
Return the phase a.k.a. argument of a complex signal.

6.10.2 Complex Conjugate

`csignal`	`cc`	`(const csignal &Input)`
`dcsignal`	`cc`	`(const dcsignal &Input)`

DESCRIPTION:
Return the complex conjugate of the input.

6.10.3 Cosine

signal	`cos`	`(const` *signal* `&Input)`

DESCRIPTION:
Return the cosine of the signal.

6.10.4 Complex Exponential

`csignal`	`euler`	`(const fsignal &Input)`
`dcsignal`	`euler`	`(const dsignal &Input)`

DESCRIPTION:
Return the complex exponential of a real signal, i.e. $e^{(ix)}$.

6.10.5 Exponential

signal	`exp`	`(const` *signal* `&Input)`

DESCRIPTION:
Return the exponential of a signal.

6.10.6 Magnitude

`fsignal`	`fabs`	`(const fsignal &Input)`
`fsignal`	`fabs`	`(const csignal &Input)`
`dsignal`	`fabs`	`(const dsignal &Input)`
`dsignal`	`fabs`	`(const dcsignal &Input)`

DESCRIPTION:
Return the magnitude of a signal, i.e. $|x|$.

6.10.7 Magnitude and Phase to Real and Imaginary Conversion

```
csignal        mptori        (const fsignal &Magnitude,
                              const fsignal &Phase)
dcsignal       mptori        (const dsignal &Magnitude,
                              const dsignal &Phase)
```

DESCRIPTION:
Convert two real signals representing the magnitude and phase of a complex signal to a complex signal in real and imaginary format.
Given real signals M and θ, `mptori`, creates a complex signal, z, by

$$\begin{aligned} z &= Me^{i\theta} \\ &= M\cos(\theta) + iM\sin(\theta) \end{aligned}$$

6.10.8 Intensity

```
fsignal        intensity     (const fsignal &Input)
fsignal        intensity     (const csignal &Input)
dsignal        intensity     (const dsignal &Input)
dsignal        intensity     (const dcsignal &Input)
```

DESCRIPTION:
Return the intensity of a signal, i.e. $|x|^2$.

6.10.9 Imaginary Part of a Complex Signal

```
fsignal        imag          (const csignal &Input)
dsignal        imag          (const dcsignal &Input)
```

DESCRIPTION:
Return the imaginary part of a complex signal.

6.10.10 Logarithms

signal	log	(const *signal* &Input)
signal	log10	(const *signal* &Input)

DESCRIPTION:
Return either the natural or base 10 logarithm of a signal.

6.10.11 Signal to a Power

signal	pow	(const *signal* &Input, const double p)

DESCRIPTION:
Raise the signal to the power p.

6.10.12 Real Part of a Complex Signal

fsignal	real	(const csignal &Input)
dsignal	real	(const dcsignal &Input)

DESCRIPTION:
Return the real part of a complex signal.

6.10.13 Sine

signal	`sin`	`(const` *signal* `&Input)`

DESCRIPTION:
Return the sine of the signal.

6.10.14 Square Root

signal	`sqrt`	`(const` *signal* `&Input)`

DESCRIPTION:
Return the square root of a signal.

7 Signal Class Functions

7.1 Autocorrelate

fsignal	autocorrelate	(const fsignal &Input)
csignal	autocorrelate	(const csignal &Input)

DESCRIPTION:
Autocorrelation function. This function is implemented as a multiplication in the Fourier domain.

EXAMPLE:

```
fsignal x , Rx ;
x  = "data" ;
Rx = autocorrelate( x ) ;
```

7.2 Average

signal	average	(const *signal* &Input, const int Nrows, const int Ncols = 1)

DESCRIPTION:
Average the rows and columns of a 2D SIGNAL.
The arguments are:

Nrows	The number of rows to average.
Ncols	The number of columns to average. If not specified, no column average is performed.

7.3 Barrel Shift

signal	bshift	(const *signal* &Input, const int Nx = 0, const int Ny = 0)

DESCRIPTION:
Barrel shift a 1D signal by Nx elements or 2D signal by Nx elements and Ny elements. If neither Nx nor Ny are specified, the signal is shifted to its center.
The arguments are:

Nx	X-shift in pixels.
Ny	Y-shift in pixels.

7.4 Clip

signal	clip	(const *signal* &Input, const *stype* Threshold)
signal	clip	(const *signal* &Input, const *stype* Threshold, const *stype* Value)

DESCRIPTION:
Clip a signal. The first form clips the data by setting each element whose value exceeds Threshold to the value Threshold. The second form replaces those elements whose value exceed the threshold by Value.
The arguments are:

Threshold	Value at which the data should be clipped.
Value	Optional value which should replace the clipped elements.

7.5 Central Moments

`double`	`cmoment`	`(const` *stype* `&Input,` `const int Order)`

`double`	`cmoment`	`(const` *stype* `&Input,`
		`const int Order,`
		`const double Mu)`

DESCRIPTION:
Compute the central moment of specified order of a 1D signal. The first form returns the central moment of specified order. The second form returns the central moment of specified order using the pre-computed center of mass `Mu`. This form works faster than the first since the center of mass does not need to be re-computed. The nth central moment of a signal $f(n)$ is defined as

$$\mu_n = \frac{1}{M}\sum_{n=0}^{N-1}(x_0 + n\Delta x - \mu_1)^n\,|f(n)|$$

where M is the mass of the signal, and μ_1 is the center of mass, computed as:

$$\begin{aligned} M &= \sum_{n=0}^{N-1}|f(n)| \\ \mu_1 &= \frac{1}{M}\sum_{n=0}^{N-1}(x_0 + n\Delta x)\,|f(n)| \\ &= \frac{x_0}{M} + \frac{\Delta x}{M}\sum_{n=0}^{N-1} n\,|f(n)| \end{aligned}$$

The arguments are:

`Order`	Order of the central moment.
`Mu`	Pre-computed center of mass.

EXAMPLE:

This code sequence:

```
int n, N = 10000 ;
fsignal
     g(1, N);
float
     x0 = -10,
     dx = 2*fabs(x0)/N,
     x ;

/*
 * Create a gaussian signal
 */
for( n=0 ; n<N ; n++ )
{
     x    = x0 + n*dx ;
     g[n] = exp( -x*x/2. ) ;
```

```
}

/*
 * Set the origin and interval
 */
originof(g, X_AXIS) = x0 ;
intervalof(g, X_AXIS) = dx ;

/*
 * Normalize the signal
 */
g /= sqrt(2*Pi) ;

save( g , "gauss1d" ) ;

fprintf(stdout,"First 5 moments of a gaussian:\n") ;
for( n=0 ; n<5 ; n++ )
     fprintf(stdout,"   %2d) %g\n",n,cmoment(g,n)) ;
```

produces

```
First 5 moments of a gaussian:
    0) 500
    1) -1.21872e-09
    2) 1
    3) 3.98998e-09
    4) 3
```

7.6 Center of Mass

`Array<double>`	`com`	`(const` *signal* `&Input)`

DESCRIPTION:
Compute the center of mass of a signal. The function returns an `Array` of `MAXDIM` elements containing the coordinates of the center of mass.

7.7 Convolve

signal	`convolve`	`(const` *signal* `&X, const` *signal* `&Kernel)`

DESCRIPTION:
Convolve two signals. The convolution is performed in the time domain.

7.8 Correlate

signal	`correlate`	`(const` *signal* `&X, const` *signal* `&Kernel)`

DESCRIPTION:
Correlate two signals. The convolution is performed in the time domain.

7.9 Crosscorrelate

signal	crosscorrelate	(const *signal* &S1, const *signal* &S2)

DESCRIPTION:
Correlate two signals. The correlation is performed by a multiplication in the Fourier domain.

7.10 Demean

signal	demean	(const *signal* &Input)
signal	demean	(const *signal* &Input, const int Axis)
int	ipdemean	(*signal* &Input)
int	ipdemean	(*signal* &Input, const int Axis)

DESCRIPTION:
Demean a signal. The first form removes the mean from the entire signal. The second form removes the mean along the specified axis of a 2D signal. `Axis` is either `X_AXIS` or `Y_AXIS`.
The `ipdemean` functions perform in-place demeans over-writing the original data. They can be used to conserve memory. They return `SUCCESS` if no errors occur and `FAILURE` otherwise.

7.11 Detrend

fsignal	detrend	(const fsignal &Input, const int Order)

DESCRIPTION:
Perform a polynomial detrend of specified order.

7.12 Differentiate

```
signal        differentiate    (const signal &Input,
                                const int Axis = X_AXIS)
```

DESCRIPTION:
Differentiate a signal. For 2D signals, the differentiation axis must be specified and defaults to the X-axis if not set. This function implements a central difference approximation to the first derivative:

$$f'(0) = \frac{-3f(0)+4f(1)-f(2)}{2\Delta x} \quad \text{For the first point}$$

$$f'(n) = \frac{f(n+1)-f(n-1)}{2\Delta x} \quad \text{For } 0 < n < N-1$$

$$f'(N-1) = \frac{3f(N-1)-4f(N-2)+f(N-3)}{2\Delta x} \quad \text{For the last point}$$

Axis is either **X_AXIS** or **Y_AXIS**.

7.13 Extract

```
signal      extract     (const signal &Input,
                         const int *Location,
                         const int *Size )
signal      extract     (const signal &Input,
                         const int Xstart,
                         const int YstartOrXsize,
                         const int ZstartOrXsize= -1,
                         const int XsizeOrYsize= -1,
                         const int Ysize= -1,
                         const int Zsize= -1 )
```

DESCRIPTION:
Extract a subset of a signal. The first form specifies the extraction origin and sizes in `int` arrays, the second form selects the origins and sizes separately.
The arguments are:

`Xstart`	The X start of the extraction.
`YstartOrXsize`	For 1D signals this specifies the size of the extraction region. For 2D or 3D signals this specifies the Y start of the extraction region.
`ZstartOrXsize`	For 2D signals this specifies the x size of the extraction region. For 3D signals this specifies the Z start of the extraction region.
`XsizeOrYsize`	For 2D signals this specifies the Ysize of the extraction region. For 3D signals this specifies the X size of the extraction region.
`YSize`	Y size of the extraction for 3D signals.
`Zsize`	Z size of the extraction for 3D signals.
`Location`	Array having at least the same number of elements as the signal argument specifying the origin of the extraction.
`Size`	Array having at least the same number of elements as the signal argument specifying the size of the extraction.

7.14 Histogram

```
fsignal        histogram        (const fsignal &Input,
                                 const int Bins = 256)

dsignal        histogram        (const dsignal &Input,
                                 const int Bins = 256)
```

DESCRIPTION:
Compute the histogram of a signal. `Bins` optionally specifies the number of bins to use. The default is 256.

7.15 Insert

```
int            insert           (signal &Base,
                                 const signal &Input,
                                 const int Xlocation = 0,
                                 const int Ylocation = 0,
                                 const int Zlocation = 0)

int            insert           (signal &Base,
                                 const signal &Input,
                                 const int *Location)
```

DESCRIPTION:
Insert one signal into another over-writing the data. The first form specifies the insertion location indices individually. If none of the locations are specified, the insertion starts at element zero. The second for specifies the insertion location in the array `Location`. This function returns `SUCCESS` upon successful insertion and `FAILURE` otherwise. `SUCCESS` and `FAILURE` are described in the subsection on Global Constants, Section 13. Note: In contrast to the `overlay` function of Section 7.26, `insert` over-writes the data in the base signal. The arguments are:

`Xlocation`	X location of the insertion.
`Ylocation`	Y location of the insertion.
`Zlocation`	Z location of the insertion.
`Location`	Array having at least the same number of elements as the base signal specifying the origin of the insertion.

7.16 Integrate

signal	`integrate`	`(const` *signal* `&Input)`

DESCRIPTION:
Integrate a signal.

7.17 Local Mean

signal	localmean	(const *signal* &Input, const int Xsize, const int Ysize = 0)

DESCRIPTION:
Compute the local mean of a 1- or 2D signal.
The arguments are:

Xsize	X size of the region.
Ysize	Y size of the region.

EXAMPLE:
This code sequence was used to create the data for Figure 1:

```
int
     N = 201,
     width,
     n ;

String Name("tower") ;

fsignal
     tower(1,N),
     lm ;

width = N/2 ;
for( n=0 ; n<5 ; n++ )
{
     tower += mkbox(N,width) ;
     width /= 2 ;
}

/*
 * Compute local mean
 */
width *= 5 ;
lm = localmean(tower,width) ;

save( tower , Name ) ;
Name += String("_lm") + String(width) ;
save( lm , Name ) ;
```

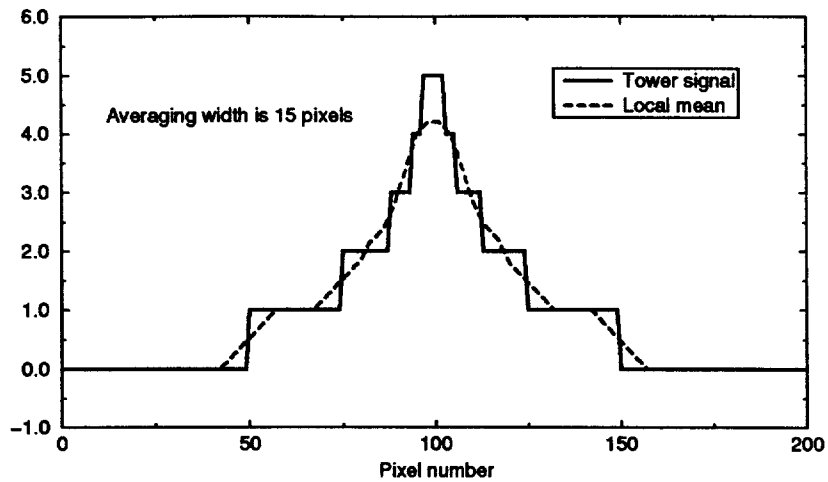

Figure 1: Stair step tower signal and its 15 pixel width local mean.

This function can be used to perform a local mean normalization:

```
/*
 * Read in a signal
 */
fsignal x = "image";

/*
 * Perform a 10x5 local mean normalization
 */
x /= localmean( x , 10 , 5 ) ;

save( x , "image_lmn" ) ;
```

7.18 Local Statistics

```
int          localstat      (const fsignal &Input,
                             fsignal &Mean,
                             fsignal &Variance,
                             const int Xsize,
                             const int Ysize = 0)

int          localstat      (const dsignal &Input,
                             dsignal &Mean,
                             dsignal &Variance,
                             const int Xsize,
                             const int Ysize = 0)

int          localstat      (const csignal &Input,
                             csignal &Mean,
                             fsignal &Variance,
                             const int Xsize,
                             const int Ysize = 0)

int          localstat      (const dcsignal &Input,
                             dcsignal &Mean,
                             dsignal &Variance,
                             const int Xsize,
                             const int Ysize = 0)
```

DESCRIPTION:
Compute the local mean and variance of a 1- or 2D signal. This function returns SUCCESS upon successful computation and FAILURE otherwise. SUCCESS and FAILURE are described in the subsection on Global Constants, Section 13.
The arguments are:

Mean	The local mean signal.
Variance	The local variance signal.
Xsize	X size of the region.
Ysize	Y size of the region.

7.19 Magnify

signal	magnify	(const *signal* &Input, const float Magnification, const int Method = R_BILINEAR)
signal	magnify	(const *signal* &Input, const float *Magnification, const int Method = R_BILINEAR)

DESCRIPTION:
Magnify the input signal using linear, bilinear or spline interpolation. Allows the signal to be changed to any arbitrary size. New sizes are specified by means of magnification(s). `Magnification` is the magnification factor. A negative magnification reduces the size of the signal, a positive magnification enlarges the signal. The first form applies the same magnification to all dimensions.
See also the `resample()` function in Section 7.31.
The arguments are:

`Magnification`	The magnification factor(s).
`Method`	Resample method. Must be one of `R_BILINEAR`, `R_SPLINE`, `R_REPLICATE`, or `R_ZEROINS`. These constants are described in Global Constants, Section 13.

7.20 Merge

fsignal	merge	(const fsignal &A, const fsignal&B)
csignal	merge	(const csignal &A, const csignal&B)

DESCRIPTION:
Merge two 1D signals by interleaving their values.

7.21 First Two Moments of a Signal

double	moments	(const *stype* &Input, Array<double> &COM)

DESCRIPTION:
Compute the center of mass and standard deviation of a signal. This function returns the variance
For a signal, $f(\vec{n})$, where $\vec{n} = (k, l, m, n)$ represents the signal indices, the center of mass, $\vec{\mu}_1$, and variance, μ_2^2, are computed as:

$$
\begin{aligned}
M &= \sum_{\vec{n}}^{\vec{N}-1} |f(\vec{n})| \\
\vec{\mu}_1 &= \frac{1}{M} \sum_{\vec{n}}^{\vec{N}-1} \vec{x}\, |f(\vec{n})| \\
&= \frac{1}{M} \sum_{\vec{n}}^{\vec{N}-1} (x_0 + k\Delta x, y_0 + l\Delta y, \cdots)\, |f(\vec{n})| \\
&= \left(x_0 + \frac{\Delta x}{M} \sum_{\vec{n}}^{\vec{N}-1} k\, |f(\vec{n})|, y_0 + \frac{\Delta y}{M} \sum_{\vec{n}}^{\vec{N}-1} l\, |f(\vec{n})|, \cdots \right) \\
&= (\mu_{1x}, \mu_{1y}, \cdots) \\
\mu_2^2 &= \frac{1}{M} \sum_{\vec{n}}^{\vec{N}-1} |\vec{x} - \vec{\mu}_1|^2\, |f(\vec{n})| \\
&= \frac{1}{M} \sum_{\vec{n}}^{\vec{N}-1} |(x_0 + k\Delta x - \mu_{1x}, y_0 + l\Delta y - \mu_{1y}, \cdots)|^2\, |f(\vec{n})| \\
&= (x_0 - \mu_{1x})^2 + 2\frac{\Delta x}{M} (x_0 - \mu_{1x}) \sum_{\vec{n}}^{\vec{N}-1} l\, |f(\vec{n})| + \frac{(\Delta x)^2}{M} \sum_{\vec{n}}^{\vec{N}-1} l^2\, |f(\vec{n})| + \\
&\quad (y_0 - \mu_{1y})^2 + 2\frac{\Delta y}{M} (y_0 - \mu_{1y}) \sum_{\vec{n}}^{\vec{N}-1} k\, |f(\vec{n})| + \frac{(\Delta y)^2}{M} \sum_{\vec{n}}^{\vec{N}-1} k^2\, |f(\vec{n})| + \cdots
\end{aligned}
$$

The arguments are:

COM	An Array of at least size ndimof(Input) of the center of mass coordinates.

7.22 Flatten a Signal to 1 Dimension

signal	flatten	(const *signal* &Input)
void	ipflatten	(*signal* &Input)

DESCRIPTION:
Flatten a signal to 1 dimension. If the input has `NDim` dimensions with elements `Dim[ NDim ]`, then `flatten` will return a 1 dimensional signal with

```
Dim[0]*···*Dim[NDim-1]
```

elements.

7.23 Fold a 1 Dimensional Signal to an N-Dimensional Signal

signal	fold	(const *signal* &Input, const int Xdim, const int Ydim, const int Zdim = 0, const int Tdim = 0)
void	ipfold	(*signal* &Input, const int Xdim, const int Ydim, const int Zdim = 0, const int Tdim = 0)

DESCRIPTION:
Fold a 1-dimensional signal to be an N-dimensional signal with specified dimensions.
The total number of elements in the signal must be the product of the dimensions.

7.24 Normalize

signal	normalize	(const *signal* &Input, const int Method = NORM_TO_MAX)
int	ipnormalize	(*signal* &Input, const int Method = NORM_TO_MAX)

DESCRIPTION:
Normalize a signal. The result will be normalized to the magnitude of the maximum value unless `Method` is set to one of `NORM_TO_SUM`, `NORM_TO_VAR`, `NORM_TO_SUMFABS`, `NORM_TO_MAXMAG`, or `NORM_TO_MEAN`.
`ipnormalize()` performs an in-place normalization in order to save memory. It returns `SUCCESS` upon successful normalization and `FAILURE` otherwise.
These normalization methods are:

Method	Explanation
NORM_TO_SUM	Normalize to magnitude of signal sum, $\lvert\sum_n x(n)\rvert$.
NORM_TO_VAR	Normalize signal to have unit variance.
NORM_TO_MAX	Normalize to magnitude of maximum value, $\lvert\max_n x(n)\rvert$. Default.
NORM_TO_SUMFABS	Normalize to sum of signal magnitudes, $\sum_n \lvert x(n)\rvert$.
NORM_TO_MAXMAG	Normalize to maximum of magnitudes, $\max_n \lvert x(n)\rvert$.
NORM_TO_MEAN	Normalize signal to have unit mean.

These constants are described in Section 13. Figure 2 compares the results of the various normalizations on a random signal. The code sequence used to generate the data is:

```
fsignal uniform( uniformrandom( -0.25 , 1 , 110 ) ) ;

save( uniform , "uniform" ) ;

save( normalize( uniform , NORM_TO_MAX )     , "uniform_nM" ) ;
save( normalize( uniform , NORM_TO_SUM )     , "uniform_ns" ) ;
save( normalize( uniform , NORM_TO_VAR )     , "uniform_nv" ) ;
save( normalize( uniform , NORM_TO_SUMFABS ), "uniform_nf" ) ;
save( normalize( uniform , NORM_TO_MAXMAG ) , "uniform_nm" ) ;
```

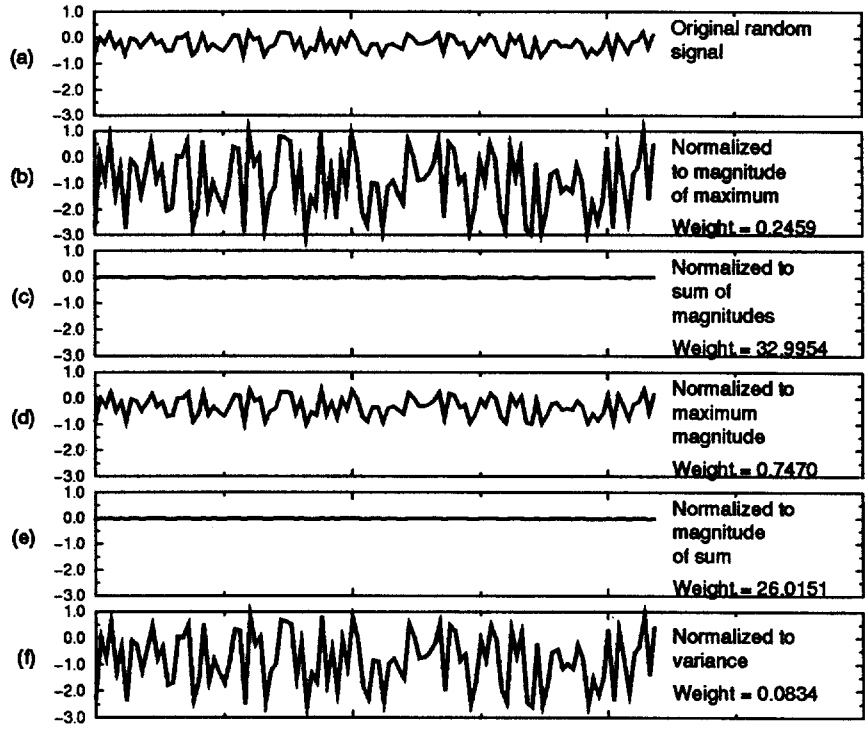

Figure 2: Effect of various normalization methods on a random signal. Graph (a) shows the original signal. Graphs (b) through (f) show the signal normalized using the method indicated along with the weight used in the normalization. All graphs are shown on the same scale for comparison.

7.25 Higher Order Approximation to First Derivative

signal	`ndifferentiate(const` *signal* `&Input, const int Order)`

DESCRIPTION:
This function implements higher order approximations (up to 6) to the first derivative. `Order` is the desired order of the approximation where

$$2 \leq \texttt{Order} \leq 6$$

7.26 Overlay

signal	`overlay`	`(const` *signal* `&Base,`
		`const` *signal* `&Input,`
		`const int Xlocation,`
		`const int Ylocation = 0,`
		`const int Zlocation = 0)`

signal	`overlay`	`(const` *signal* `&Base,`
		`const` *signal* `&Input,`
		`const int *Location)`

DESCRIPTION:
Overlay one signal into another returning a third. The first form specifies the overlay origins individually. The second form specifies the overlay location in the array `Location`. The signal arguments remain unchanged. Note: In contrast to the **insert** function of Section 7.15, `overlay` does not over-write the data in the base signal but rather returns a third.
The arguments are:

`Location`	Array having at least the same number of elements as the base signal specifying the origin of the overlay.
`Xlocation`	X location of the insertion.
`Ylocation`	Y location of the insertion.
`Zlocation`	Z location of the insertion.

7.27 Probability Density Function

These three functions estimate the probability density function (PDF) of a signal. The first two use a rectangular Parzen [6] window technique which is equivalent to smoothing the histogram with a rectangular window. As a consequence, signals whose PDFs have step discontinuities in them such as uniform and exponential distributions, will not be correctly estimated around the discontinuities.
Figures 3, 4, and 5 compare the results of the three functions for a uniformly distributed, gaussian distributed, and exponentially distributed signal, respectively. The uniform and exponential cases demonstrate how the Parzen technique does not accurately estimate the distribution around the step discontinuities.
The code sequence used to generate the data for the figures is:

```
int N = 1000 ;

fsignal
     u( uniformrandom(2, 1, N) ),
     g( gaussianrandom(2, 2, N) ),
     e( exprandom(1, N) );

save( u      , "u" ) ;
save( pdf(u) , "u_pdf" ) ;
save( pdfi(u), "u_pdfi" ) ;
save( pdfh(u), "u_pdfh" ) ;

save( g      , "g" ) ;
save( pdf(g) , "g_pdf" ) ;
save( pdfi(g), "g_pdfi" ) ;
save( pdfh(g), "g_pdfh" ) ;

save( e      , "e" ) ;
save( pdf(e) , "e_pdf" ) ;
save( pdfi(e), "e_pdfi" ) ;
save( pdfh(e), "e_pdfh" ) ;
```

7.27.1 PDF

```
fsignal        pdf            (const fsignal &Input,
                               const float Percent = 0.10,
                               const int Bins = 512 )
```

DESCRIPTION:
This function estimates the pdf using a rectangular Parzen [6] window technique. Note: It does not fill in zero bins.
The arguments are:

`Percent`	Percentage of data range the rectangular window covers.
`Bins`	Number of bins to use when quantizing the data.

7.27.2 Interpolated PDF

```
fsignal        pdfi           (const fsignal &Input,
                               const float Percent = 0.10,
                               const int Bins = 512 )
```

DESCRIPTION:
This function estimates the pdf using a rectangular Parzen [6] window technique and then uses cubic spline interpolation to fill in zero bins.
The arguments are:

`Percent`	Percentage of data range the rectangular window covers.
`Bins`	Number of bins to use when quantizing the data.

7.27.3 PDF from Histogram

```
fsignal        pdfh          (const fsignal &Input,
                              const int Bins = 512 )
```

DESCRIPTION:
This function estimates the pdf by normalizing the histogram by the signal sample interval times the signal sum.
The arguments are:

Bins	Number of bins to use when quantizing the data.

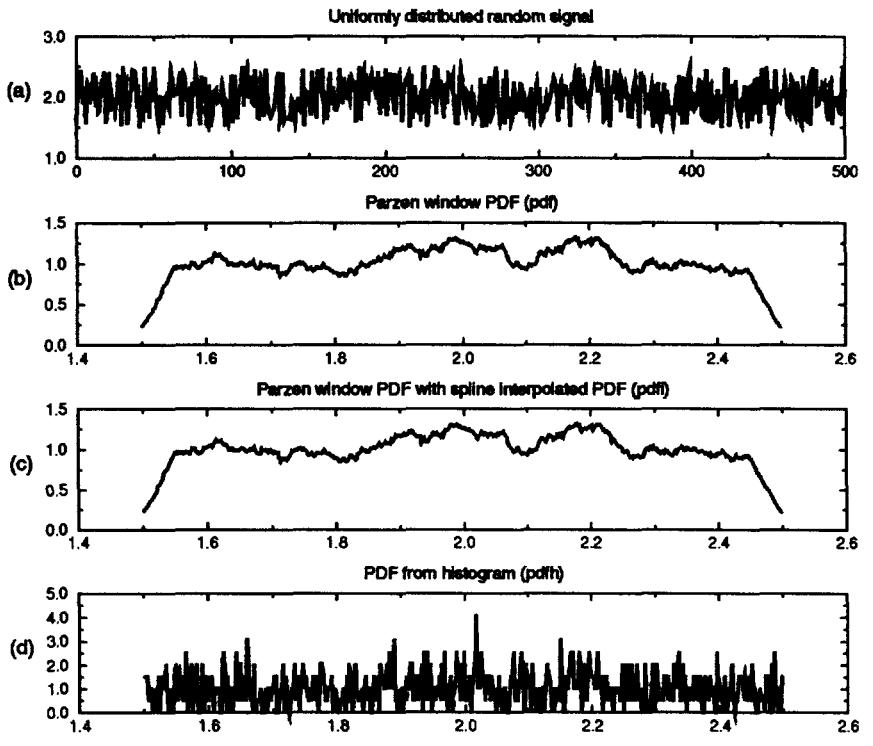

Figure 3: Comparison of the three PDF estimators on a uniformly distributed random signal. Graph (a) shows the original signal. Graphs (b) through (d) show the signal show the estimated PDF. Note: The Parzen window technique does not perform well around the edges of the distribution due to its averaging nature.

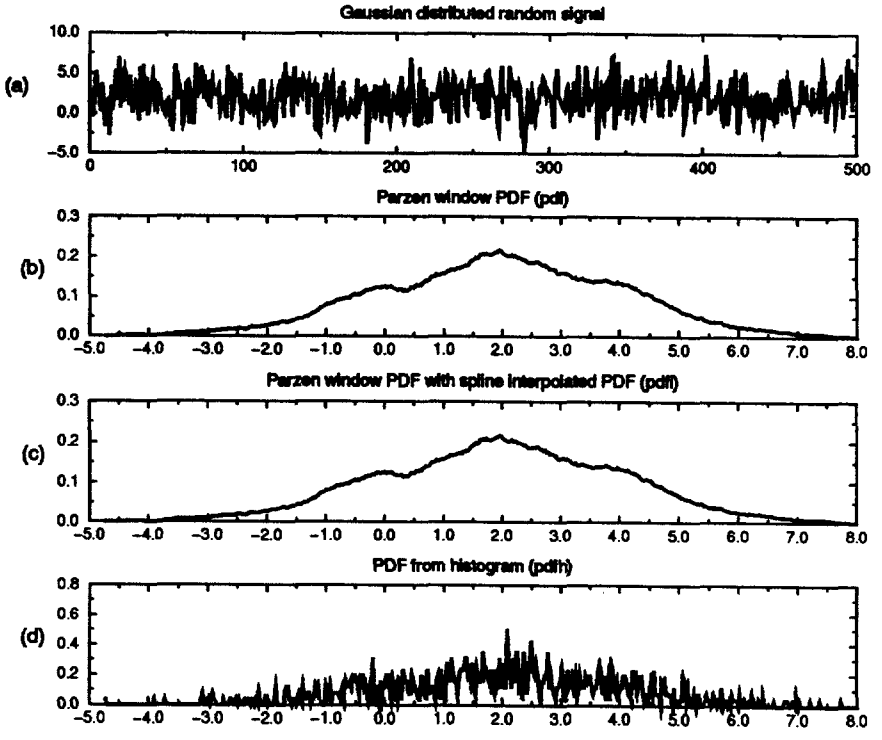

Figure 4: Comparison of the three PDF estimators on a gaussian distributed random signal. Graph (a) shows the original signal. Graphs (b) through (d) show the signal show the estimated PDF.

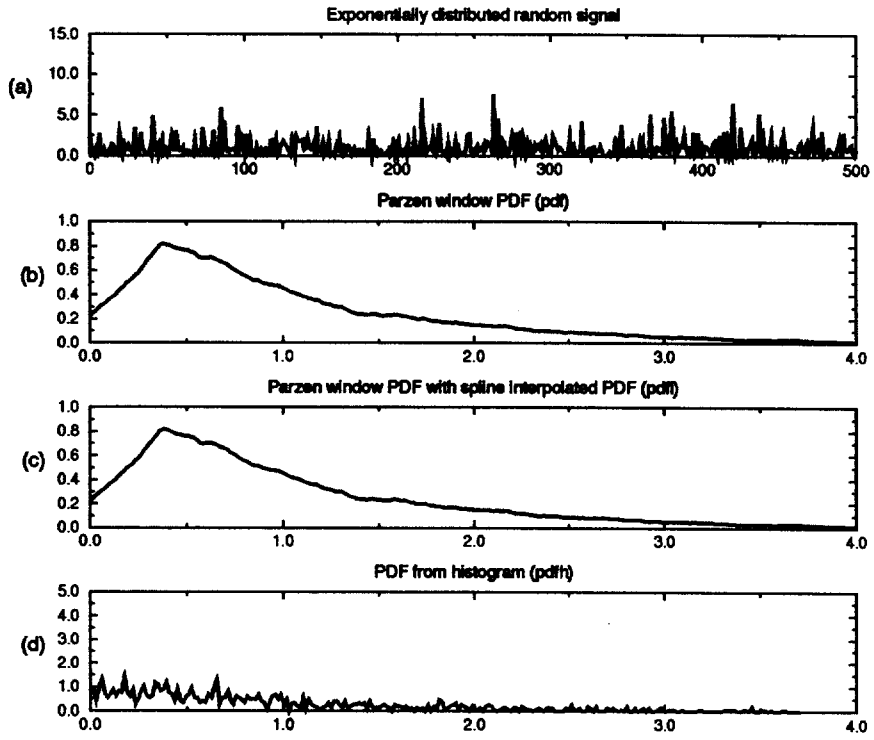

Figure 5: Comparison of the three PDF estimators on an exponentially distributed random signal. Graph (a) shows the original signal. Graphs (b) through (d) show the signal show the estimated PDF. Note: The Parzen window technique does not perform well around the discontinuity at the origin due to its averaging nature.

7.28 Powerspectrum

`fsignal`	`powerspectrum`	`(const fsignal &Input)`

DESCRIPTION:
Compute the powerspectrum of the argument. If $R = |\mathcal{F}(x)|^2$, where $\mathcal{F}(\cdot)$ is the Fourier transform of the sequence, then the function returns $R/$`nelemof`(R).

7.29 Project

signal	`project`	`(const` *signal* `&Input,` `const int Axis = Y_AXIS)`

DESCRIPTION:
Project a 2D signal down to a 1D signal by summing along the specified axis. By default the projection axis is the Y-axis.
The result can be normalized by dividing by the axis dimension:

```
fsignal a( "test" ) , a_proj ;
a_proj = project( a , X_AXIS ) / dimof( a , X_AXIS ) ;
```

7.30 Replicate

signal	`replicate`	`(const` *signal* `&Input,`
		`const int N,`
		`const int Axis)`

DESCRIPTION:
Create a two dimensional signal from a one dimensional signal by replicating it `N` times along the specified axis.
The arguments are:

`Input`	Source signal. It can be either a one dimensional signal or a two dimensional signal with one of its dimensions as 1.
`N`	Number of times to replicate the input.
`Axis`	Axis along which to replicate. Must be either `X_AXIS` or `Y_AXIS`.

7.31 Resample

```
signal      resample    (const signal &Input,
                         const int NewDim,
                         const int Method = R_BILINEAR)
signal      resample    (const signal &Input,
                         const int *NewDim,
                         const int Method = R_BILINEAR)
```

DESCRIPTION:
Resample the input signal using linear, bilinear or spline interpolation for 1-, 2- or 3-D signals. Allows the signal to be changed to any arbitrary size. New sizes are specified by `NewDim`. The first form resamples the input so that all dimensions have size `NewDim`. The second form resamples the signal to the sizes specified in the array `NewDim`.
See also the `magnify()` function in Section 7.19.
The arguments are:

`NewDim`	The new signal dimensions.
`Method`	Resample method. Must be one of `R_BILINEAR`, `R_SPLINE`, `R_REPLICATE`, or `R_ZEROINS`. These constants are described in Global Constants, Section 13.

7.32 Rotate

```
signal      rotate      (const signal &Input,
                         const float Angle)
signal      rotate      (const signal &Input,
                         const float Angle,
                         const int Xcenter,
                         const int Ycenter)
```

DESCRIPTION:
Rotate a signal. The first form rotates the input about its center. The second form rotates the input about the specified center.
The arguments are:

`Angle`	Rotation angle in degrees.
`Xcenter`	X location of the center of rotation.
`Ycenter`	Y location of the center of rotation.

7.33 Shift

signal	`shift`	`(const` *signal* `&Input,`
		`const int XShift,`
		`const int YShift = 0)`

DESCRIPTION:
Shift a signal. Values shifted in are zeroed. Values shifted out are lost.

7.34 Smooth

signal	`smooth`	`(const` *signal* `&Input,`
		`const int Xsize,`
		`const int Ysize = 0)`

DESCRIPTION:
Smooth a signal by averaging with a moving rectangular window.
The arguments are:

`Xsize`	X-dimension of the smoothing window.
`Ysize`	Y-dimension of the smoothing window.

7.35 Sum Magnitudes and Intensities of Signal Elements

double	sumfabs	(const *signal* &Input)
double	sumint	(const *signal* &Input)

DESCRIPTION:
sumfabs and **sumint** sum magnitudes and intensities (magnitude squared) of signal elements, respectively:

$$\texttt{sumfabs(x)} = \sum_{n=0}^{N-1} |x(n)|$$

$$\texttt{sumint(x)} = \sum_{n=0}^{N-1} |x(n)|^2$$

7.36 SNR

```
float          snr          (const fsignal &Input,
                             const fsignal &Template,
                             const int Verbose = 0,
                             const char *Basename = 0)
```

DESCRIPTION:
Estimate the signal-to-noise ratio (snr) of a signal using a specified template. The function implements:

$$\text{snr} = \frac{(\max(R_{it}) - \langle R_{nt} \rangle)^2}{\text{Var}(R_{NT})}$$

where R_{it} is the cross-correlation between the input and the template; and R_{nt} is the cross-correlation between the estimate of the noise and the template.
The returned snr is in dB.
The arguments are:

Verbose	Put the function in verbose mode. The function will report when either the input signal or template need to be zero padded so they have identical dimensions. Additionally, $\max(R_{it})$, $< R_{nt} >$, and $\text{Var}(R_{NT})$ will be written to stdout.
Basename	Base name under which intermediary signals should be saved. If set, the shifted templated will be saved as Basename_Ts, the cross-correlation between the image and template as Basename_Rit, the template autocorrelation as Basename_Rtt, and the noise estimate as Basename_NEst.

EXAMPLE:

This code segment:

```
int N = 10000 ;
float
     sd = 2 ,
     m0 = 0 ,
     m1 = m0 + 5*sd ,
     var, mean ;

fsignal
     temp(  gaussianrandom(m1, sd, N) ) ,
     noise( gaussianrandom(m0, sd, N) ) ,
     signal( temp + noise ) ;

/*
 * Verify the statistics of the two gaussians
 */
statistics(temp,&mean,&var) ;
fprintf(stdout,"Template: mean=%7.4f, var=%7.4f\n",mean,var) ;

statistics(noise,&mean,&var) ;
fprintf(stdout,"Noise    : mean=%7.4f, var=%7.4f\n\n",mean,var) ;

/*
```

```
     * Estimated SNR
     */
    float SNR = snr( signal, temp, 1) ;

    /*
     * Actual SNR
     */
    SNR = 10*log10( N*pow(m1-m0,2)/(sd*sd) ) ;
    fprintf(stdout,"Actual SNR = %5.2f dB\n",SNR) ;
```

will produce

```
Template: mean=10.0065, var= 4.0533
Noise   : mean= 0.0396, var= 4.0413

Ritmax =  225.35
Rntmean=    1.78e-09
Rntvar =    1.52
SNR    =   45.25 dB
Actual SNR = 53.98 dB
```

7.37 Spectrogram

```
csignal        spectrogram       (const fsignal &Input,
                                  const int WindowSize,
                                  const int WindowStep)

csignal        spectrogram       (const csignal &Input,
                                  const int WindowSize,
                                  const int WindowStep)
```

DESCRIPTION:
Compute the spectrogram or windowed Fourier transform of a 1D signal.
The arguments are:

`WindowSize`	Size of the window.
`WindowStep`	Step size by which the window location is incremented.

7.38 Statistics

```
void        statistics    (const fsignal &Input,
                           float *Mean,
                           float *Variance,
                           const int IgnoreZeros = 0)

void        statistics    (const dsignal &Input,
                           double *Mean,
                           double *Variance,
                           const int IgnoreZeros = 0)

void        statistics    (const csignal &Input,
                           complex *Mean,
                           float *Variance,
                           const int IgnoreZeros = 0)

void        statistics    (const dcsignal &Input,
                           dcomplex *Mean,
                           double *Variance,
                           const int IgnoreZeros = 0)
```

DESCRIPTION:
Compute the mean and variance of a signal.
Let $f(n)$ be an N-element signal. The mean is given by

$$\begin{aligned} \mu &= \frac{1}{N}\sum_{n=0}^{N-1} f(n) \\ &= \langle f(n) \rangle \end{aligned}$$

The variance is given by

$$\begin{aligned} \sigma^2 &= \frac{1}{N}\sum_{n=0}^{N-1} (f(n)-\mu)^2 \\ &= \frac{1}{N}\sum_{n=0}^{N-1} f^2(n) - \mu^2 \\ &= \langle f^2(n) \rangle - \mu^2 \quad . \end{aligned}$$

The arguments are:

`Mean`	The signal mean.
`Variance`	The signal variance.
`IgnoreZeros`	If set, zero data values will be left out of the calculations.

EXAMPLE:
This code sequence:

```
int N = 20000 ;
fsignal
     u( uniformrandom(10, 1, N) ),
     g( gaussianrandom(2, 2, N) );
csignal
     z( g, g ) ;
float mean , var ;
complex zmean ;

statistics( u , &mean , &var ) ;
fprintf(stdout,"Uniformly distributed random:\n") ;
fprintf(stdout,"\tmean=%8.5f, var=%8.5f, sd=%8.5f\n\n",
        mean,var,sqrt(var)) ;

statistics( g , &mean , &var ) ;
fprintf(stdout,"Gaussian distributed random:\n") ;
fprintf(stdout,"\tmean=%8.5f, var=%8.5f, sd=%8.5f\n\n",
        mean,var,sqrt(var)) ;

statistics( z , &zmean , &var ) ;
fprintf(stdout,"Complex gaussian distributed random:\n") ;
fprintf(stdout,"\tmean=(%8.5f,%8.5f), var=%8.5f, sd=%8.5f\n",
        real(zmean),imag(zmean),var,sqrt(var)) ;
```

produces:

```
Uniformly distributed random:
        mean= 9.99987, var= 0.08268, sd= 0.28754

Gaussian distributed random:
        mean= 2.01567, var= 4.07046, sd= 2.01754

Complex gaussian distributed random:
        mean=( 2.01567, 2.01567), var= 8.14093, sd= 2.85323
```

7.39 Threshold

```
signal        threshold        (const signal &Input,
                                const stype Threshold,
                                const stype Value = 0.0 )
```

DESCRIPTION:
Threshold a signal by setting each element whose value is less than `Threshold` to `Value`.
The arguments are:

`Threshold`	Value at which the data should be thresholded.
`Value`	Optional value which should replace the thresholded elements.

7.40 Fourier Transforms

IDP++ offers the following fast Fourier transforms:

- Real to complex with optional half-plane transform `fft`;
- Complex to complex `fft`;
- Half-plane complex to real `hfft`;
- Real to complex row transform of two dimensional data sets with optional half-plane transform `row_fft`;
- Real to complex column transform of two dimensional data sets with optional half-plane transform `col_fft`;
- Complex to complex row transform of two dimensional data sets `row_fft`;
- Complex to complex column transform of two dimensional data sets `col_fft`;
- Half-plane complex to real row transform of two dimensional data sets `row_hfft`;
- Half-plane complex to complex column transform of two dimensional data sets `col_hfft`;

All of these functions implement a fast Fourier transform using Lincoln Laboratory's `llfft` code [8] with some modifications by Lawrence Livermore National Laboratory. Optionally, they offer control over the sign of the transform, sign modulation [9] a.k.a. "toggling" to move the transform domain origin to the center of the data set, and scaling.

Notes:

- The scaling on the transform is $1/\sqrt{N}$ where N is the total number of points in the transformed data set;
- These functions implicitly zero-pad data sets whose sizes are not powers of two to the nearest power of two.
- There are two sets of FFT functions which perform identical tasks but have different calling sequences. The first set take only two arguments: An input signal and a logical OR code sequence used to specify the operation of the transform function. The second set take a long list arguments to control the operation.

  The codes used in the first set of FFT functions are

CODE	MEANING
FFT_TB	Toggle input data set before transform.
FFT_TA	Toggle input data set after transform.
FFT_HP	Set half plane transform.
	A real input will result in a complex half-plane output.
	A complex input will result in a real output.
FFT_NM	Normalize result.
FFT_SP	Set transform sign positive.
FFT_SN	Set transform sign negative.

For example,

```
fsignal Input ;
csignal Result ;
   ...
Result = fft( Input , FFT_TB | FFT_SP | FFT_HP ) ;
   ...
```

would result in a half-plane, positive sign transform with the input data set being toggled before the FFT. Alternatively, you could write

```
fsignal Input ;
csignal Result ;
int code = FFT_TB | FFT_SP | FFT_HP ;
   ...
Result = fft( Input , code ) ;
   ...
```

7.40.1 Column Fourier Transforms

```
csignal        col_fft        (const fsignal &Input,
                               const Code = FFT_SN|FFT_NM)

csignal        col_fft        (const csignal &Input,
                               const Code = FFT_SN|FFT_NM)

dcsignal       col_fft        (const dsignal &Input,
                               const Code = FFT_SN|FFT_NM)

dcsignal       col_fft        (const dcsignal &Input,
                               const Code = FFT_SN|FFT_NM)
```

```
csignal        col_fft        (fsignal &Input,
                               int Sign,
                               int PreToggle,
                               int PostToggle,
                               int HalfPlane,
                               int Normalize)

csignal        col_fft        (csignal &Input,
                               int Sign,
                               int PreToggle,
                               int PostToggle,
                               int HalfPlane,
                               int Normalize)

dcsignal       col_fft        (dsignal &Input,
                               int Sign,
                               int PreToggle,
                               int PostToggle,
                               int HalfPlane,
                               int Normalize)

dcsignal       col_fft        (dsignal &Input,
                               int Sign,
                               int PreToggle,
                               int PostToggle,
                               int HalfPlane,
                               int Normalize)
```

DESCRIPTION:
Compute the Fourier transform of each of the columns of a two dimensional signal.

The arguments are:

`Code`	Operation code.
`Sign`	The sign of the transform. This is the sign of i in the exponent.
`PreToggle`	If set to 1, the data are sign modulated (toggled) before transforming to move the origin in the transform to the center of the data set.
`PostToggle`	If set to 1, the data are sign modulated (toggled) after transforming to restore the data set.
`HalfPlane`	When set with a real input, a half-plane transform is returned. When set with a complex input, this indicates the data represent a half-plane transform.
`Normalize`	When set, the transform is normalized.

7.40.2 Half-Plane Column Fourier Transforms

```
fsignal        col_hfft        (const csignal &Input,
                                 const Code = FFT_SP|FFT_NM)

dsignal        col_hfft        (const dcsignal &Input,
                                 const Code = FFT_SP|FFT_NM)
```

```
fsignal        col_hfft        (csignal &Input ,
                                 int Sign,
                                 int PreToggle,
                                 int PostToggle,
                                 int Normalize)

dsignal        col_hfft        (dcsignal &Input ,
                                 int Sign,
                                 int PreToggle,
                                 int PostToggle,
                                 int Normalize)
```

DESCRIPTION:
Compute the inverse Fourier transform of a the columns of a two dimensional complex signal representing a half-plane transforms.
The arguments are:

`Code`	Operation code.
`Sign`	The sign of the transform. This is the sign of i in the exponent.
`PreToggle`	If set to 1, the data are sign modulated (toggled) before transforming to move the origin in the transform to the center of the data set.
`PostToggle`	If set to 1, the data are sign modulated (toggled) after transforming to restore the data set.
`Normalize`	When set, the transform is normalized by $1/\sqrt{N}$.

7.40.3 Fourier Transform

```
csignal        fft        (const fsignal &Input,
                           const Code = FFT_SN|FFT_NM)

csignal        fft        (const csignal &Input,
                           const Code = FFT_SP|FFT_NM)

dcsignal       fft        (const dsignal &Input,
                           const Code = FFT_SN|FFT_NM)

dcsignal       fft        (const dcsignal &Input,
                           const Code = FFT_SP|FFT_NM)
```

```
csignal        fft        (fsignal &Input,
                           int Sign,
                           int PreToggle,
                           int PostToggle,
                           int HalfPlane,
                           int Normalize)

csignal        fft        (csignal &Input,
                           int Sign,
                           int PreToggle,
                           int PostToggle,
                           int Normalize)

dcsignal       fft        (dsignal &Input,
                           int Sign,
                           int PreToggle,
                           int PostToggle,
                           int HalfPlane,
                           int Normalize)

dcsignal       fft        (dcsignal &Input,
                           int Sign,
                           int PreToggle,
                           int PostToggle,
                           int Normalize)
```

DESCRIPTION:
Compute the Fourier transform of a signal.
The arguments are:

`Code`	Operation code.
`Sign`	The sign of the transform. This is the sign of i in the exponent.
`PreToggle`	If set to 1, the data are sign modulated (toggled) before transforming to move the origin in the transform to the center of the data set.
`PostToggle`	If set to 1, the data are sign modulated (toggled) after transforming to restore the data set.
`HalfPlane`	When set with a real input, a half-plane transform is returned. When set with a complex input, this indicates the data represent a half-plane transform.
`Normalize`	When set, the transform is normalized by $1/\sqrt{N}$.

7.40.4 Half-Plane Fourier Transforms

```
fsignal        hfft        (const csignal &Input,
                            const Code = FFT_SP|FFT_NM)

dsignal        hfft        (const dcsignal &Input,
                            const Code = FFT_SP|FFT_NM)
```

```
fsignal        hfft        (csignal &Input,
                            int Sign,
                            int PreToggle,
                            int PostToggle,
                            int Normalize)

dsignal        hfft        (dcsignal &Input,
                            int Sign,
                            int PreToggle,
                            int PostToggle,
                            int Normalize)
```

DESCRIPTION:
Compute the inverse Fourier transform of a complex signal representing a half-plane transform.
The arguments are:

`Code`	Operation code.
`Sign`	The sign of the transform. This is the sign of i in the exponent.
`PreToggle`	If set to 1, the data are sign modulated (toggled) before transforming to move the origin in the transform to the center of the data set.
`PostToggle`	If set to 1, the data are sign modulated (toggled) after transforming to restore the data set.
`Normalize`	When set, the transform is normalized by $1/\sqrt{N}$.

7.40.5 Row Fourier Transforms

```
csignal        row_fft        (const fsignal &Input,
                               const Code = FFT_SN|FFT_NM)

csignal        row_fft        (const csignal &Input,
                               const Code = FFT_SN|FFT_NM)

dcsignal       row_fft        (const dsignal &Input,
                               const Code = FFT_SN|FFT_NM)

dcsignal       row_fft        (const dcsignal &Input,
                               const Code = FFT_SN|FFT_NM)
```

```
csignal        row_fft        (fsignal &Input,
                               int Sign,
                               int PreToggle,
                               int PostToggle,
                               int HalfPlane,
                               int Normalize)

csignal        row_fft        (csignal &Input,
                               int Sign,
                               int PreToggle,
                               int PostToggle,
                               int HalfPlane,
                               int Normalize)

dcsignal       row_fft        (dsignal &Input,
                               int Sign,
                               int PreToggle,
                               int PostToggle,
                               int HalfPlane,
                               int Normalize)

dcsignal       row_fft        (dcsignal &Input,
                               int Sign,
                               int PreToggle,
                               int PostToggle,
                               int HalfPlane,
                               int Normalize)
```

DESCRIPTION:
Compute the Fourier transform of each of the rows of a two dimensional signal.

The arguments are:

`Code`	Operation code.
`Sign`	The sign of the transform. This is the sign of i in the exponent.
`PreToggle`	If set to 1, the data are sign modulated (toggled) before transforming to move the origin in the transform to the center of the data set.
`PostToggle`	If set to 1, the data are sign modulated (toggled) after transforming to restore the data set.
`HalfPlane`	When set with a real input, a half-plane transform is returned. When set with a complex input, this indicates the data represent a half-plane transform.
`Normalize`	When set, the transform is normalized by $1/\sqrt{N}$.

7.40.6 Half-Plane Row Fourier Transforms

```
fsignal        row_hfft        (const csignal &Input,
                                 const Code = FFT_SP|FFT_NM)

dsignal        row_hfft        (const dcsignal &Input,
                                 const Code = FFT_SP|FFT_NM)
```

```
fsignal        row_hfft        (csignal &Input ,
                                 int Sign,
                                 int PreToggle,
                                 int PostToggle,
                                 int Normalize)

dsignal        row_hfft        (dcsignal &Input ,
                                 int Sign,
                                 int PreToggle,
                                 int PostToggle,
                                 int Normalize)
```

DESCRIPTION:
Compute the inverse Fourier transform of the rows of a two dimensional complex signal representing half-plane transforms.
The arguments are:

`Code`	Operation code.
`Sign`	The sign of the transform. This is the sign of i in the exponent.
`PreToggle`	If set to 1, the data are sign modulated (toggled) before transforming to move the origin in the transform to the center of the data set.
`PostToggle`	If set to 1, the data are sign modulated (toggled) after transforming to restore the data set.
`Normalize`	When set, the transform is normalized by $1/\sqrt{N}$.

7.41 Order Statistic Filters

Order statistic filters ([4], [5]) operate on ordered sub-sets of the data. Given a set of data points, $x_i, i = 0, 1, \ldots, n-1$, the i-th order statistic, $x_{(i)}$, is the i-th element after the set of data points have been ordered from lowest to highest in value.
The order statistic filters in IDP++ operate on regions of the data, replacing the center pixel by the selected order statistic of the region.
There are five order statistic filters:

- MIN;
- MAX;
- Median;
- α-Trimmed Mean;
- Edge-Trimmed Mean.

The MIN and MAX filters select the lowest, $x_{(0)}$, and the highest, $x_{(n-1)}$ order statistic respectively.
The trimming filters perform arithmetic averages of sub-sets of the ordered statistics. They require a parameter specifying the percentage of points to trim out.

7.41.1 α-Trimmed Mean

```
fsignal        atmean        (const fsignal &Input,
                              const float alpha,
                              const int XRegion ,
                              const int YRegion = 0)
dsignal        atmean        (const dsignal &Input,
                              const double alpha,
                              const int XRegion ,
                              const int YRegion = 0)
```

DESCRIPTION:
Perform an α-trimmed mean of the signal. The α-trimmed mean of a set of n values is:

$$\overline{x}_\alpha = \frac{1}{(1-2\alpha)n} \sum_{i=\alpha n}^{(1-\alpha)n-1} x_{(i)} \qquad 0 \leq \alpha \leq 0.5$$

α selects the percentage of order statistics at each end of the set to eliminate from the average. Note: For $\alpha = 0$, $\overline{x}_\alpha$ reduces to the arithmetic mean, and as $\alpha \to 0.5$, i.e. 50% trimming from each end, $\overline{x}_\alpha \to \text{median}(x_i)$.

7.41.2 Edge-Trimmed Mean

```
fsignal        etmean        (const fsignal &Input,
                              const float alpha,
                              const int XRegion ,
                              const int YRegion = 0)
dsignal        etmean        (const dsignal &Input,
                              const double alpha,
                              const int XRegion ,
                              const int YRegion = 0)
```

DESCRIPTION:
Perform an edge-trimmed mean of the signal. The edge-trimmed mean of a set of n values is:

$$\overline{x}_\alpha = \begin{cases} -\frac{1}{\alpha n} \sum_{i=-\alpha n}^{n-1} x_{(i)} & -1 \leq \alpha < 0 \\ \frac{1}{\alpha n} \sum_{i=0}^{\alpha n-1} x_{(i)} & 0 < \alpha \leq 1 \end{cases}$$

The edge-trimmed mean filter eliminates a percentage of data points from one or the other edge of the distribution. α selects the percentage. As a matter of bookkeeping, negative α trims the low end of the distribution, and positive α trims the high end. Note: For $\alpha = 1$, i.e. 100%, $\overline{x}_\alpha$ reduces to the arithmetic mean.

7.41.3 MAX Filter

```
fsignal        MAX           (const fsignal &Input,
                              const int XRegion ,
                              const int YRegion = 0)
dsignal        MAX           (const dsignal &Input,
                              const int XRegion ,
                              const int YRegion = 0)
```

DESCRIPTION:
Perform a MAX filter of the data set. The MAX filter selects the highest order statistic of the region, $x_{(n-1)}$.

7.41.4 Median Filter

```
fsignal        median        (const fsignal &Input,
                              const int XRegion ,
                              const int YRegion = 0)
dsignal        median        (const dsignal &Input,
                              const int XRegion ,
                              const int YRegion = 0);
```

DESCRIPTION:
Perform a median filter of the signal. The median of a set of n values is:

$$\mathrm{med}\,(x_i) = \begin{cases} x_{(\nu)} & n = 2\nu + 1 \\ \frac{1}{2}\left(x_{(\nu-1)} + x_{(\nu)}\right) & n = 2\nu \end{cases}$$

The median selects the middle value or average of the two middle values if the number of points is even.

7.41.5 MIN Filter

```
fsignal        MIN           (const fsignal &Input,
                              const int XRegion ,
                              const int YRegion = 0)
dsignal        MIN           (const dsignal &Input,
                              const int XRegion ,
                              const int YRegion = 0)
```

DESCRIPTION:
Perform a MIN filter of the data set. The MIN filter selects the lowest order statistic of the region, $x_{(0)}$.

7.42 Signal Padding

These functions zero-pad signals in various manners:

- Right or right and bottom padding;
- Center padding;
- Zero-pad with wrapping (1D only).

7.42.1 Center Padding

```
signal      centerpad     (const signal &Input ,
                           const int xsize ,
                           const int ysize = 0,
                           const int zsize = 0,
                           const int tsize = 0)

signal      centerpad     (const signal &Input, const int *Dim)
```

DESCRIPTION:
Center-pad a signal, i.e. pad around its edges to the specified dimensions.
The first form accepts the new dimensions separately.
The second form accepts the new dimensions in the array `Dim` which must be of at least size `ndimof(Input)`.

7.42.2 Wrap-Around Padding

```
signal      wrappad       (const signal &Input,
                           const int xsize,
                           const int xstart)
```

DESCRIPTION:
Zero-pad a 1D signal to size `xsize` starting at element `xstart` wrapping around to the beginning if `dimof(Input,X_AXIS)` > `xsize`.

7.42.3 Zero Padding

```
signal      zeropad     (const signal &Input,
                         const int xsize ,
                         const int ysize = 0,
                         const int zsize = 0,
                         const int tsize = 0)

signal      zeropad     (const signal &Input, const int *Dim)

signal      zeropad     (const signal &Input,
                         const int *Dim,
                         const int *Location)
```

DESCRIPTION:
Zero-pad a signal.
The first form accepts the new dimensions separately and pads the right or right and bottom.
The second form accepts the new dimensions in the array `Dim` which must be of at least size `ndimof(Input)` and pads the right or right and bottom.
The third form accepts a starting location and pads around the input.

7.43 Spline Interpolation

These functions perform cubic spline interpolations, and compute spline coefficients and subsequently evaluate them.
The coefficients b_i, c_i, and d_i, $i = 0, 1, \ldots, N-1$ are computed for a cubic interpolating spline

$$s(x) = y_i + (x - x_i)b_i + (x - x_i)^2 c_i + (x - x_i)^3 d_i$$

for

$$x_i \leq x \leq x_{i+1}$$

N is the number of data points or knots ($n \geq 2$), x_i are the abscissas of the knots in strictly increasing order, and y_i are the ordinates of the knots.
b_i, c_i, and d_i are the spline coefficients as defined above.

7.43.1 Evaluate Spline

These functions provide three methods for evaluating splines once the coefficients have been computed. See Section 12.9 for a single point evaluation.

```
float          evaluate        (const float xvalue,
                                 const int N,
                                 const float *x,
                                 const float *y,
                                 const float *b,
                                 const float *c,
                                 const float *d)

double         evaluate        (const double xvalue,
                                 const int N,
                                 const double *x,
                                 const double *y,
                                 const double *b,
                                 const double *c,
                                 const double *d)
```

DESCRIPTION:
Given an array of X, Y pairs, and spline coefficients, compute the value of interpolation at the specified point `xvalue`.

`xvalue`	X-value where interpolation is performed.
`N`	Size of data arrays
`x`	Pointer to the existing X-values. These are the knots of the spline.
`y`	Pointer to the existing Y-values.
`b`	Pointer to b coefficients.
`c`	Pointer to c coefficients.
`d`	Pointer to d coefficients.

```
float          evaluate        (const float xvalue,
                                 const fsignal &X,
                                 const fsignal &Y,
                                 const fsignal &Coefficients)

double         evaluate        (const double xvalue,
                                 const dsignal &X,
                                 const dsignal &Y,
                                 const dsignal &Coefficients)
```

```
fsignal        evaluate        (const fsignal &Xvalues,
                                const fsignal &X,
                                const fsignal &Y,
                                const fsignal &Coefficients)

dsignal        evaluate        (const dsignal &Xvalues,
                                const dsignal &X,
                                const dsignal &Y,
                                const dsignal &Coefficients)
```

DESCRIPTION:
The **evaluate** function evaluates a spline at a specified value or sequence of values given the knots and spline coefficients. The coefficients are returned by the **spline** function (Section 7.43.3).

xvalue	X-value where the spline is to be evaluated.
Xvalues	Signal of X-values where the spline is to be evaluated.
X	Signal of existing X-values. These are the knots of the spline.
Y	Signal of existing Y-values.
Coefficients	Signal of b, c, d, coefficients as returned by the **spline** function.

7.43.2 Interpolate

```
fsignal        interpolate      (const fsignal &X,
                                 const fsignal &Y,
                                 const int N,
                                 const float Origin,
                                 const float Interval)
dsignal        interpolate      (const dsignal &X,
                                 const dsignal &Y,
                                 const int N,
                                 const float Origin,
                                 const float Interval)
```

```
fsignal        interpolate      (const float *X,
                                 const float *Y,
                                 const int Npts,
                                 const int N,
                                 const float Origin,
                                 const float Interval)
dsignal        interpolate      (const double *X,
                                 const double *Y,
                                 const int Npts,
                                 const int N,
                                 const float Origin,
                                 const float Interval)
```

DESCRIPTION:
Given a list of knots, either as signals or arrays, an origin, an interval, and a signal size, return a signal interpolated at fixed intervals from the origin.
Thus if x_0 and Δx are a data origin and interval, respectively, create a signal which is interpolated using the knots by successively evaluating a spline at $x_0 + n\Delta x$, $n = 0, 1, \ldots, N-1$.

`X`	X-values where the spline is to be evaluated.
`Y`	Y-values.
`Npts`	Size of input arrays (if not signals).
`N`	Size of returned signal.
`Origin`	Origin of returned signal.
`Interval`	Sample interval of returned signal.

7.43.3 Computation of Spline Coefficients

```
fsignal        spline        (const float *x,
                              const float *y,
                              const int N)
dsignal        spline        (const double *x,
                              const double *y,
                              const int N)
```

DESCRIPTION:
Given a list of X, Y pairs, return a 2D, `N` by 3, signal of spline coefficients.

```
fsignal        spline        (const fsignal &X, const fsignal &Y)
dsignal        spline        (const dsignal &X, const dsignal &Y)
```

DESCRIPTION:
Given a list of X, Y pairs, return a 2D `N` by 3 signal of spline coefficients.

```
fsignal        spline        (const fsignal &Input,
                              fsignal &X, fsignal &Y)
dsignal        spline        (const dsignal &Input,
                              dsignal &X, dsignal &Y)
```

DESCRIPTION:
Compute spline coefficients for an input signal.

7.44 Random Number Generators

7.44.1 Class A Random Numbers

```
fsignal        classArandom      (const float A,
                                  const float Gamma,
                                  const int XDim)
fsignal        classArandom      (const float A,
                                  const float Gamma,
                                  const float Sigma,
                                  const int XDim)
```

DESCRIPTION:
This function returns a field of Class A [7] random numbers generated by the Two-Term Approximation (TTA) to the Class A distribution.
The first form uses the two-parameter TTA:

$$w(x \mid A, \Gamma') = \frac{e^{-A}}{\sqrt{2\pi}\,\sigma_1} e^{-\frac{x^2}{2\sigma_1^2}} + \frac{1-e^{-A}}{\sqrt{2\pi}\,\sigma_2} e^{-\frac{x^2}{2\sigma_2^2}}$$

where

$$\sigma_1^2 = \frac{\Gamma'}{\Gamma'+1}$$

$$\sigma_2^2 = \frac{A\Gamma'+1}{A(\Gamma'+1)}$$

This two-sided distribution has zero mean. Its variance is:

$$\sigma^2 = \frac{A\Gamma'+1-e^{-A}}{A(\Gamma'+1)}$$

The second form uses the three-parameter form:

$$w(x \mid A, \Gamma', \sigma_g) = \frac{e^{-A}}{\sqrt{2\pi}\,\sigma_g} e^{-\frac{x^2}{2\sigma_g^2}} + \frac{1-e^{-A}}{\sqrt{2\pi}\,\sigma_g\, c} e^{-\frac{x^2}{2\sigma_g^2 c^2}}$$

where

$$\begin{aligned} \sigma_g^2 &= \frac{\Gamma'}{\Gamma'+1} \\ &= \sigma_1^2 \\ c^2 &= 1 + \frac{1}{A\Gamma'} \end{aligned}$$

The variance of the three-parameter form is:

$$\sigma^2 = \sigma_g^2 \left(e^{-A} + (1-e^{-A})c^2\right)$$

Note:

$$\begin{aligned} \lim_{A\to 0} w(x \mid A, \Gamma', \sigma_g) &= \mathcal{N}(0, \sigma_g) \\ \lim_{A\to \infty} w(x \mid A, \Gamma', \sigma_g) &= \mathcal{N}(0, c\sigma_g) \\ \lim_{\Gamma'\to \infty} w(x \mid A, \Gamma', \sigma_g) &= \mathcal{N}(0, 1) \end{aligned}$$

where $\mathcal{N}(0, \sigma_g)$ is a zero-mean Gaussian with standard deviation of σ_g

7.44.2 Exponentially Distributed Random Numbers

fsignal	exprandom	(const float theta, const int N)
fsignal	exprandom	(const fsignal& theta)

DESCRIPTION:
Return a signal of exponentially distributed random numbers. The first form uses the scalar theta as the global mean. The second form uses the signal theta as point means. Figure 6 shows a sample data set with estimated probability density function.
The functional form of the distribution is

$$w(x \mid \theta) = \frac{1}{\theta} e^{-x/\theta}$$

with mean and variance

$$\begin{aligned} \mu &= \theta \\ \sigma^2 &= \theta^2. \end{aligned}$$

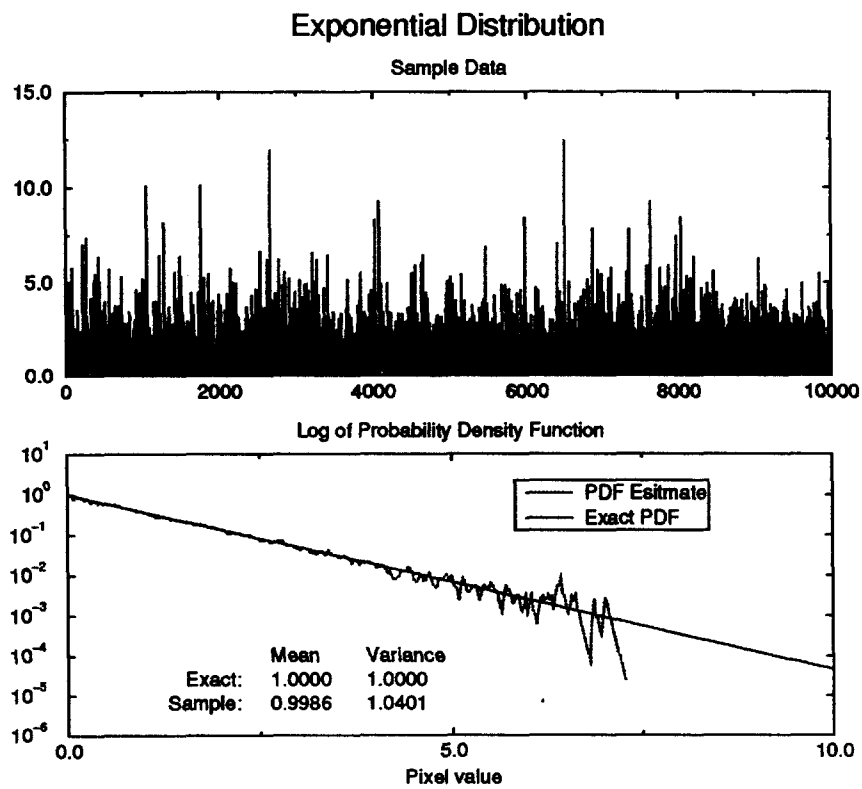

Figure 6: Sample exponentially-distributed random data set.

7.44.3 Gamma Random Numbers

```
float       gammarandom     (const float Alpha,
                             const float Beta,
                             const int *seed)

fsignal     gammarandom     (const float Alpha,
                             const float Beta,
                             const int NDim,
                             const int *Dim)

fsignal     gammarandom     (const float Alpha,
                             const float Beta,
                             const int Nx,
                             const int Ny = 0,
                             const int Nz = 0,
                             const int Nt = 0)
```

DESCRIPTION:
Compute a gamma-distributed random variable. The distribution is

$$f(x) = \frac{1}{\Gamma(\alpha)\ \beta^{\alpha}} x^{\alpha-1} e^{-x/\beta} \quad x > 0$$

The mean is $\alpha\beta$ and the variance is $\alpha\beta^2$.
The first form returns a single random number. The second form returns a signal of random numbers. Figure 7 shows a sample data set with estimated probability density function.
The arguments are:

`Alpha`	The alpha parameter of the distribution.
`Beta`	The beta parameter of the distribution.
`NDim`	The dimensionality of the result.
`Dim`	An array of size of at least `NDim` containing the dimensions of the result.
`Nx`	X-dimension of the result.
`Ny`	Y-dimension of the result.
`Nz`	Z-dimension of the result.
`Nt`	t-dimension of the result.
`seed`	Optional seed to randomize the generation. Must be less than zero.

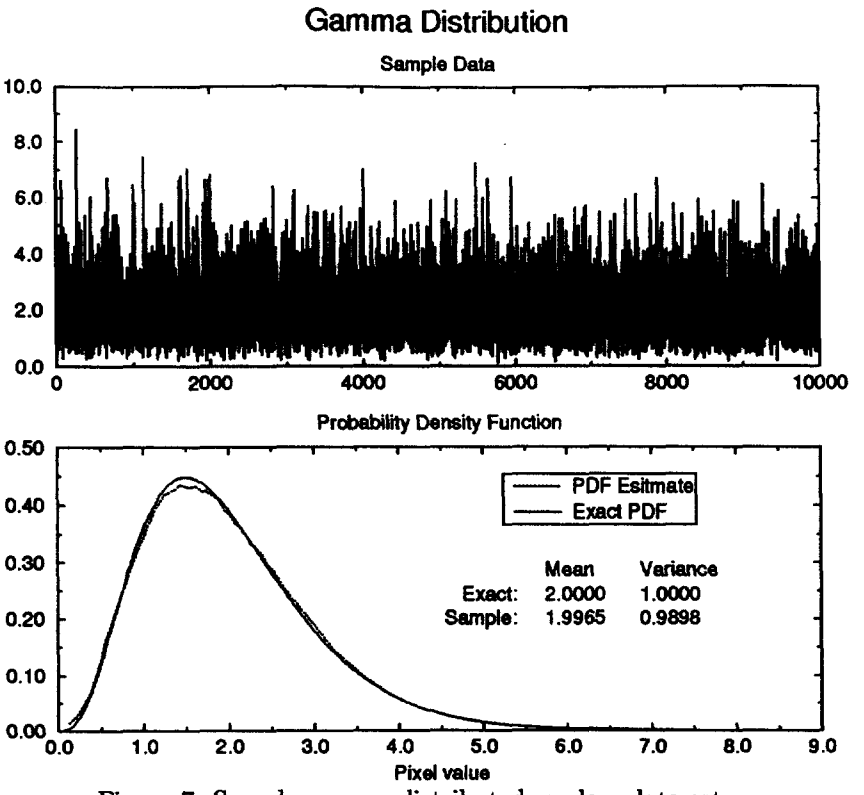

Figure 7: Sample gamma-distributed random data set.

7.44.4 Gaussian Random Numbers

```
double      gaussianrandom   (int *seed)

float       gaussianrandom   (const float Mean,
                              const float SD, int *seed)

fsignal     gaussianrandom   (const fsignal &Mean,
                              const fsignal &SD)

fsignal     gaussianrandom   (const float Mean,
                              const float SD,
                              const int NDim,
                              const int *Dim)

fsignal     gaussianrandom   (const fsignal &Mean,
                              const float SD)

fsignal     gaussianrandom   (const float Mean,
                              const float SD,
                              const int Nx,
                              const int Ny = 0,
                              const int Nz = 0,
                              const int Nt = 0)
```

DESCRIPTION:
Generate Gaussian random variables. The first form returns a zero-mean unit-variance Gaussian random number. The second form returns a random number having the specified mean and standard deviation. The last three forms return signals whose values have the means and standard deviations specified by the `Mean` and `SD` signals respectively.

The arguments are:

`Mean`	Either a mean value or a signal specifying the means of each point.
`SD`	Either a standard deviation value or a signal specifying the standard deviations of each point.
`NDim`	The dimensionality of the result.
`Dim`	An array of size of at least `NDim` containing the dimensions of the result.
`Nx`	X-dimension of the result.
`Ny`	Y-dimension of the result.
`Nz`	Z-dimension of the result.
`Nt`	t-dimension of the result.
`seed`	Optional seed to randomize the generation. Must be less than zero.

7.44.5 K-Distributed Random Numbers

```
fsignal        Krandom        (const int Sided,
                               const float nu,
                               const float b,
                               const int N)
```

DESCRIPTION:
Return a signal of one- or two-sided K-Distributed random numbers. The choice of distribution is made by setting `Sided` to either 1 or 2. Plots of sample data sets with estimated distributions are shown in Figures 8 and 9.
The functional form of the one-sided distribution is

$$w(r \mid \nu, b) = \frac{2b^{\frac{\nu+1}{2}} r^{\frac{\nu-1}{2}}}{\Gamma(\nu)} \mathrm{K}_{\nu-1}(2\sqrt{br})$$

with mean and variance

$$\mu = \frac{\nu}{b}$$
$$\sigma^2 = \frac{\nu(\nu+2)}{b^2}$$

At the origin

$$\lim_{r\to 0} w(r \mid \nu, b) = \frac{\Gamma(\nu-1)}{\Gamma(\nu)}$$

The functional form of the two-sided distribution is

$$w(x \mid \nu, b) = \frac{2^{\frac{1}{2}(\frac{3}{2}-\nu)} b^{\frac{1}{2}(\frac{1}{2}+\nu)}}{\sqrt{\pi}\,\Gamma(\nu)} |x|^{\nu-1/2} \mathrm{K}_{\nu-\frac{1}{2}}(\sqrt{2b}\,|x|)$$

with variance

$$\sigma^2 = \frac{\nu}{b}$$

At the origin

$$\lim_{x\to 0} w(x \mid \nu, b) = \sqrt{\frac{b}{2\pi}} \frac{\Gamma(\nu-\frac{1}{2})}{\Gamma(\nu)}$$

Note that as $\nu \to \infty$, $w(x \mid \nu, b)$ approaches a zero-mean unit-variance Gaussian.

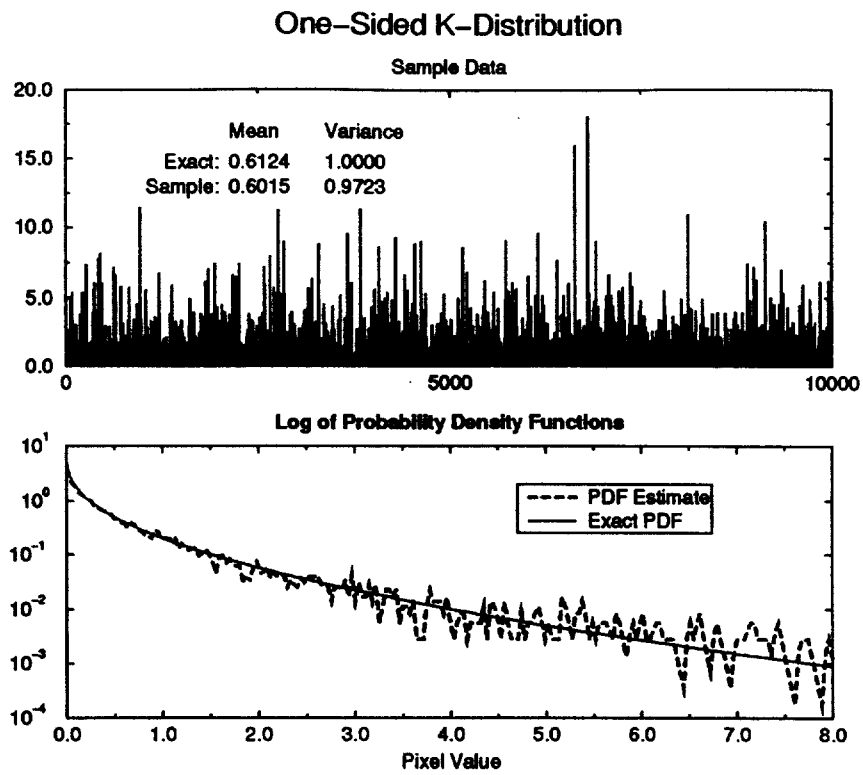

Figure 8: Sample one-sided K-distributed random data set.

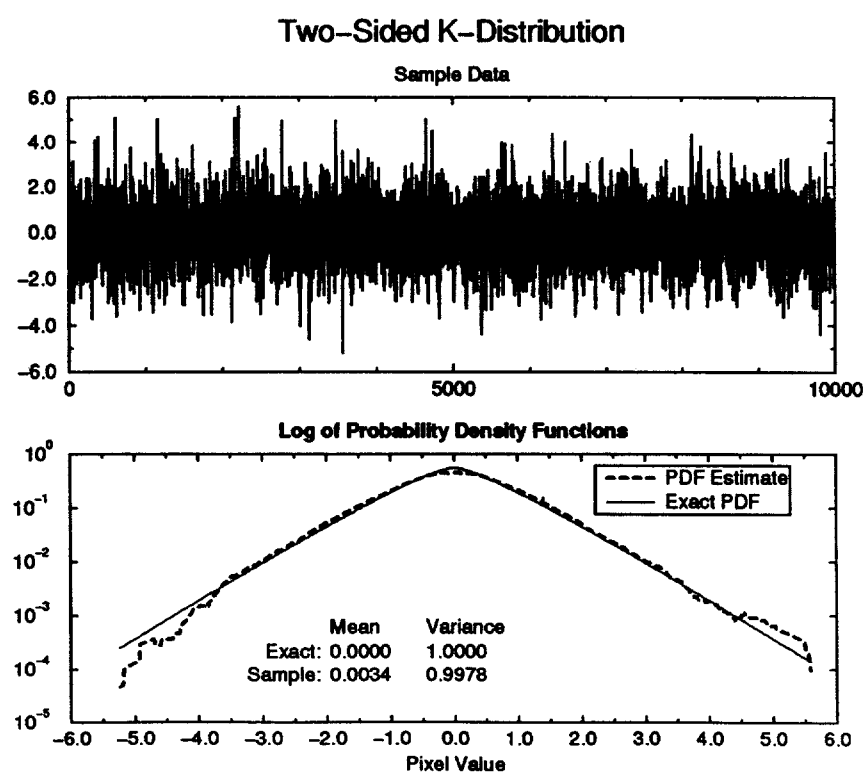

Figure 9: Sample two-sided K-distributed random data set.

7.44.6 Uniform Random Numbers

```
float        uniformrandom    (const float mean,
                               const float Range,
                               int *seed )

fsignal      uniformrandom    (const fsignal &Mean,
                               const float Range)

fsignal      uniformrandom    (const float mean,
                               const float Range,
                               const int NDim,
                               const int *Dim)

fsignal      uniformrandom    (const float mean,
                               const float Range,
                               const int Nx,
                               const int Ny = 0,
                               const int Nz = 0,
                               const int Nt = 0)
```

DESCRIPTION:
Compute a uniformly-distributed random variable. The first form returns a single random number. The second form returns a signal of random numbers using the signal `Mean` for the mean values for each point. The output numbers are generated using

$$y = (Rx) - \mu$$

where y is the returned value, R is the range, μ is the mean, and x is a random variable uniformly distributed on $[-0.5, 0.5]$.
The arguments are:

`mean`	The distribution mean.
`Range`	The distribution range.
`Mean`	Signal of means for each point.
`NDim`	Dimensionality of the result.
`Dim`	An array of size of at least `NDim` containing the dimensions of the result.
`Nx`	X-dimension of the result.
`Ny`	Y-dimension of the result.
`Nz`	Z-dimension of the result.
`Nt`	t-dimension of the result.
`seed`	Optional seed to randomize the generation. Must be less than zero. Positive seeds will be converted to negative.

7.45 Matrix Operations

The IDP++ matrix operation functions consist of:

- Vector (dot) inner products, `inner` ;
- Euclidean norms, `enorm` ;
- Compute the determinant of a matrix, `det` ;
- General "A x minus y" where A is a matrix and, x and y are vectors, `gaxmy` ;
- General "A x plus y" where A is a matrix and, x and y are vectors, `gaxpy` ;
- A function to print a signal in a matrix format, `matrixform`;
- Matrix-matrix multiply, `mmm` ;
- A transpose which returns a new signal, `transpose`, and an in-place transpose which over-writes the input in order to save memory, `iptranspose` ;
- A weighted inner product, "x-transpose A y" where A is a matrix and, x and y are vectors, `xtay`;
- Matrix inverse, `inverse`;
- Identity matrix generator, `identity`;
- Matrix equation solver, `solve`.

7.45.1 Identity Matrix

`float`	`identity`	`(const int Rows, const int Cols = 0)`

DESCRIPTION:
Generate a `Rows` by `Cols` identity matrix. If `Cols` is zero, a square `Rows` by `Rows` matrix will be created. Note, identity matrices of the three other signal types can be generated by appropriate casting:

```
dsignal  dI = dsignal( identity(4, 6) ) ;
csignal  cI = csignal( identity(10) ) ;
dcsignal zI = dcsignal( identity(5, 3) ) ;
```

7.45.2 Matrix Inverse

signal	inverse	(const *signal* &A)
signal	inverse	(const *signal* &A, const *stype* *Determinant)
int	ipinverse	(*signal* &A)
int	ipinverse	(*signal* &A, const *stype* *Determinant)

DESCRIPTION:
Return the inverse of a square matrix. These functions will optionally return the determinant if a pointer is supplied.
ipinverse performs an in-place inverse over-writing the original data in order to save memory. It returns SUCCESS upon successful inversion and FAILURE otherwise. Note: Following a failure, the original matrix data may be corrupted.
These functions call the LINPACK Netlib routines cgefa and cgedi where *c* is one of s (for float), d (for double), c (for complex), and z (for dcomplex).

7.45.3 Matrix Equation Solver

signal	solve	(const *signal* &A, const *signal* &b)
int	ipsolve	(const *signal* &A, *signal* &b)

DESCRIPTION:
Solve the system $Ax = b$.
ipsolve performs an in-place solve over-writing the original b vector data in order to save memory. It returns SUCCESS upon successful solution and FAILURE otherwise.
This function calls the LINPACK Netlib routines cgeco and cgesl where *c* is one of s (for float), d (for double), c (for complex), and z (for dcomplex).

7.45.4 Vector Inner Product

stype	`inner`	`(const` *signal* `&x, const signal &y)`

DESCRIPTION:
Vector inner products. For real signals, the result is

$$\begin{aligned} r &= \vec{x} \cdot \vec{y} \\ &= \sum_{n=0}^{N-1} x[n]y[n] \end{aligned}$$

For complex signals, the **x** signal is conjugated:

$$\begin{aligned} r &= \vec{x}^* \cdot \vec{y} \\ &= \sum_{n=0}^{N-1} x^*[n]y[n] \end{aligned}$$

7.45.5 Euclidean Norm

`double`	`enorm`	`(const` *signal* `&x)`

DESCRIPTION:
Euclidean norm:

$$\begin{aligned} r &= \|\vec{x}\| \\ &= \sqrt{\sum_{n=0}^{N-1} |x[n]|} \end{aligned}$$

7.45.6 Determinant

stype	det	(const *signal* &A)

DESCRIPTION:
Compute the determinant of the square matrix.
This function calls the LINPACK Netlib routines *c*gefa and *c*gedi where *c* is one of s (for float), d (for double), c (for complex), and z (for dcomplex).

7.45.7 GAXMY

signal	gaxmy	(*signal* &A, *signal* &x, *signal* &y)

DESCRIPTION:
General "A x minus y" where A is a matrix, and x and y are vectors.

7.45.8 GAXPY

signal	gaxpy	(*signal* &A, *signal* &x, *signal* &y)
signal	gaxpy	(*signal* &A, *signal* &x)

DESCRIPTION:
General "A x plus y" where A is a matrix, and x and y are vectors. The second form implements "A x", i.e. the case where the y vector is zero.

7.45.9 matrixform

```
void          matrixform      (const signal &A,
                               const char*format,
                               const FILE*fp=stdout)
```

DESCRIPTION:
Write a signal in a matrix form using the specified print format. The default is to write to `stdout` although any open file pointer can be used.
`format` can be any `fprintf` format but must be able to accommodate the signal's data type: real or complex.

EXAMPLE:

Let

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

$$C = \begin{bmatrix} 1+i & 2+i2 & 3+i3 \\ 4+i4 & 5+i5 & 6+i6 \\ 7+i7 & 8+i8 & 9+i9 \end{bmatrix}$$

Then if A is an `fsignal`and C a `csignal`, the following code

```
fprintf(stdout,"A is:\n") ;
matrixform( A , "%3.0f" ) ;
fprintf(stdout,"\n") ;
matrixform( C , "(%3.0f,%3.0f) " ) ;
fprintf(stdout,"C is:\n") ;
```

would produce:

```
A is:
  1  2  3  4
  5  6  7  8
  9 10 11 12

C is:
( 1, 1) ( 2, 2) ( 3, 3)
( 4, 4) ( 5, 5) ( 6, 6)
( 7, 7) ( 8, 8) ( 9, 9)
```

7.45.10 Matrix-Matrix Multiply

signal	mmm	(*signal* &A, *signal* &B)

DESCRIPTION:
Matrix-matrix multiply.

7.45.11 Transpose Functions

signal	transpose	(const *signal* &A)
int	iptranspose	(*signal* &A)

DESCRIPTION:
Transpose a signal.
transpose returns a new signal. **iptranspose** performs an in-place transpose, over-writing the original data. It returns SUCCESS upon successful completion and FAILURE otherwise.
Note, an N-element 1D signal will become a 1xN element 2D signal.
iptranspose uses the TOMS Netlib routine 513 to perform the in-place transform.

7.45.12 Weighted Inner Product

stype	`xtay`	`(const` *signal* `&x,`
		`const` *signal* `&A,`
		`const` *signal* `&y)`

stype	`xtay`	`(const` *signal* `&x,` `const` *signal* `&A)`

DESCRIPTION:
Weighted inner product. For real signals:

$$\begin{aligned} r &= \vec{x} \cdot A \cdot \vec{y} \\ &= \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x[m]A[m,n]y[n] \end{aligned}$$

For complex signals, the **x** signal is conjugated:

$$\begin{aligned} r &= \vec{x}^* \cdot A \cdot \vec{y} \\ &= \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x^*[m]A[m,n]y[n] \end{aligned}$$

The two argument form of the function implements:

$$\begin{aligned} r &= \vec{x} \cdot A \cdot \vec{x} \\ &= \sum_{m=0}^{M-1} \sum_{n=0}^{M-1} x[m]A[m,n]x[n] \end{aligned}$$

for real signals, and

$$\begin{aligned} r &= \vec{x}^* \cdot A \cdot \vec{x} \\ &= \sum_{m=0}^{M-1} \sum_{n=0}^{M-1} x^*[m]A[m,n]x[n] \end{aligned}$$

for complex

7.46 Function Generators

These functions return fields used for simulation and windows.
The signal processing window functions (`bartlett`, `blackman`, `hamming`, `hanning`, `mkbox`, `mkcircle`, `mkellipse`, and `mkgauss`) return one- or two-dimensional signals as a function of pixel value, m or (m, n).
The simulation functions (`mkchirp`, `mkcos`, `mkeuler`, `mkfield`, `mkramp`, `mksin`, and `mksqrwv`) return one- or two-dimensional fields as a function of coordinate value, $x = m\ h$, or $(x, y) = (m, n)\ h$, where h is the sample interval.
Note, throughout this section, the following notation is used:

r	is the function sample rate, uniform in x and y.
$h = 1/r$	is the sample interval, uniform in x and y.
m	is the x pixel value.
n	is the y pixel value.
$x = m\ h$	is the x coordinate value.
$y = n\ h$	is the y coordinate value.
θ	is the orientation of the wave vector in degrees.
$k_x = \cos(\pi\theta/180)$	is the x-component of the unit wave vector.
$k_y = \sin(\pi\theta/180)$	is the y-component of the unit wave vector.

7.46.1 Bartlett Window

```
fsignal        bartlett        (const int Nx, const int Ny = 0)
fsignal        bartlett        (const int NDim, const int *Dim)
```

DESCRIPTION:
Create a 1- or 2D Bartlett window. The first form requires the result dimensions to be specified separately. If `Ny` is left unspecified, a 1D signal is returned. The second form requires a dimensionality and an array of dimensions.
For a data set of size N_x, the Bartlett window is described by

$$w(m) = \begin{cases} \frac{2m}{N_x - 1} & 0 \le m \le \frac{N_x - 1}{2} \\ 2 - \frac{2m}{N_x - 1} & \frac{N_x - 1}{2} \le m \le N_x - 1 \end{cases}$$

The arguments are:

`Nx`	Size of the X-dimension in pixels.
`Ny`	Size of the Y-dimension in pixels.
`NDim`	Dimensionality of the result. This can be either 1 or 2.
`Dim`	Pointer to an `int`array of at least `NDim` elements containing the result dimensions.

7.46.2 Blackman Window

`fsignal`	`blackman`	`(const int Nx, const int Ny = 0)`
`fsignal`	`blackman`	`(const int NDim, const int *Dim)`

DESCRIPTION:
Create a 1- or 2D Blackman window. The first form requires the result dimensions to be specified separately. If `Ny` is left unspecified, a 1D signal is returned. The second form requires a dimensionality and an array of dimensions.
For a data set of size N_x, the Blackman window is described by

$$w(m) = 0.42 - 0.5\cos\left(\frac{2\pi m}{N_x - 1}\right) + 0.08\cos\left(\frac{4\pi m}{N_x - 1}\right)$$

The arguments are:

`Nx`	Size of the X-dimension in pixels.
`Ny`	Size of the Y-dimension in pixels.
`NDim`	Dimensionality of the result. This can be either 1 or 2.
`Dim`	Pointer to an `int`array of at least `NDim` elements containing the result dimensions.

7.46.3 Hamming Window

`fsignal`	`hamming`	`(const int Nx, const int Ny = 0)`
`fsignal`	`hamming`	`(const int NDim, const int *Dim)`

DESCRIPTION:
Create a 1- or 2D Hamming window. The first form requires the result dimensions to be specified separately. If **Ny** is left unspecified, a 1D signal is returned. The second form requires a dimensionality and an array of dimensions.
For a data set of size N_x, the Hamming window is described by

$$w(m) = 0.54 - 0.46 \cos\left(\frac{2\pi m}{N_x - 1}\right)$$

The arguments are:

Nx	Size of the X-dimension in pixels.
Ny	Size of the Y-dimension in pixels.
NDim	Dimensionality of the result. This can be either 1 or 2.
Dim	Pointer to an `int`array of at least **NDim** elements containing the result dimensions.

EXAMPLE:
Apply a Hamming window to taper a signal before Fourier transforming:

```
fsignal x ;
csignal x_fft ;
// x is set somewhere
x_fft = fft( x * hamming( ndimof( x ) , dimof( x ) ) ) ;
```

7.46.4 Hanning Window

fsignal	hanning	(const int Nx, const int Ny = 0)
fsignal	hanning	(const int NDim, const int *Dim)

DESCRIPTION:
Create a 1- or 2D Hanning window. The first form requires the result dimensions to be specified separately. If Ny is left unspecified, a 1D signal is returned. The second form requires a dimensionality and an array of dimensions.
For a data set of size N_x, the Hanning window is described by

$$w(m) = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi m}{N_x - 1}\right)\right]$$

The arguments are:

Nx	Size of the X-dimension in pixels.
Ny	Size of the Y-dimension in pixels.
NDim	Dimensionality of the result. This can be either 1 or 2.
Dim	Pointer to an intarray of at least NDim elements containing the result dimensions.

EXAMPLE:
Apply a Hanning window to taper a signal before Fourier transforming:

```
fsignal x ;
csignal x_fft ;
// x is set somewhere
x_fft = fft( x * hanning( ndimof( x ) , dimof( x ) ) ) ;
```

7.46.5 Make a Box

```
fsignal        mkbox        (const int NDim,
                             const int *Dim,
                             const int *BoxSize,
                             const int PutInCorner = 0)
fsignal        mkbox        (const int Nx ,
                             const int NyOrBx ,
                             const int Bx = 0 ,
                             const int By = 0 ,
                             const int PutInCorner = 0)
```

DESCRIPTION:
Create a 1- or 2D box window. The first form accepts the result and box dimensions as elements of arrays. The second form specifies the dimensions separately.
The arguments are:

`NDim`	The dimensionality of the result.
`Dim`	Array of at least size `NDim` specifying the result dimensions.
`BoxSize`	Array of at least size `NDim` specifying the box dimensions.
`Nx`	X-dimension of result.
`NyOrBx`	Y-dimension for 2D result or X-dimension of box for 1D result.
`Bx`	X-dimension of box.
`By`	Y-dimension of box.
`PutInCorner`	Specifies the box should be put in the corner of the result otherwise it is centered.

7.46.6 Make a Chirp

```
fsignal        mkchirp        (const float f0,
                               const float ChirpRate,
                               const float Rate,
                               const int NDim,
                               const int *Dim,
                               const float Orientation)
fsignal        mkchirp        (const float f0,
                               const float ChirpRate,
                               const float Rate,
                               const int Nx,
                               const int Ny = 0,
                               const float Orientation = 0)
```

DESCRIPTION:
Create a 1-, or 2D chirp signal.
If f_0 is the base frequency in Hertz, and a the chirp rate in Hertz per second, then `mkchirp` returns a sampled version of

$\sin(2\,\pi\,(f_0 + a\,x/2)\,x)$ in 1D, and
$\sin(2\,\pi\,(f_0 + a\,(k_x\,x\,+\,k_y\,y)/2)(k_x\,x\,+\,k_y\,y))$ in 2D.

The arguments are:

`f0`	Base frequency of the chirp in Hertz.
`ChirpRate`	Chirp rate in Hertz per second.
`SampleRate`	Rate at which the chirp should be sampled in Hertz.
`NDim`	The dimensionality of the result.
`Dim`	Array of at least size `NDim` specifying the result dimensions.
`Nx`	X-dimension of the result.
`Ny`	Y-dimension of the result.
`Orientation`	Applies to 2D results. Specifies the orientation in degrees of the plane wave. This is the wave vector.

7.46.7 Make a Circle or Sphere

```
fsignal        mkcircle        (const float Radius,
                                const int NDim,
                                const int *Dim)
fsignal        mkcircle        (const float Radius,
                                const int NDim,
                                const int *Dim,
                                const int *Center)
fsignal        mkcircle        (const float Radius,
                                const int Nx,
                                const int Ny,
                                const int Nz = 0)
```

DESCRIPTION:
Create a 2-, or 3D circular window.
This function will fill in an area or volume according to

$$(x - x_c)^2 + (y - y_c)^2 \quad \leq \quad 1$$

for two dimensions; and

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 \quad \leq \quad 1$$

for three; where x_c, y_c, z_c represent the pixel center of the circle.
The arguments are:

`Radius`	Radius of the window.
`NDim`	The dimensionality of the result.
`Dim`	Array of at least size `NDim` specifying the result dimensions.
`Center`	Array of at least size `NDim` specifying the location of the center of the circle or sphere.
`Nx`	X-dimension of the result.
`Ny`	Y-dimension of the result.
`Nz`	Z-dimension of the result.

7.46.8 Make a Cosine

```
fsignal        mkcos        (const float Frequency,
                             const float SampleRate,
                             const int NDim,
                             const int *Dim,
                             const float Orientation,
                             const float Phase)
fsignal        mkcos        (const float Frequency,
                             const float SampleRate,
                             const int Nx,
                             const int Ny = 0,
                             const float Orientation = 0,
                             const float Phase = 0)
```

DESCRIPTION:
Create a 1-, or 2D cosine signal.
If f is the frequency in Hertz, and ϕ the phase, then **mkcos** returns:
$\cos(2\ \pi\ f\ (x\ +\ \phi))$ in 1D, and
$\cos(2\ \pi\ f\ (k_x\ x\ + k_y\ y\ +\ \phi))$ in 2D.
The arguments are:

`Frequency`	Frequency of the cosine in Hertz.
`SampleRate`	Rate at which the cosine should be sampled in Hertz.
`NDim`	The dimensionality of the result.
`Dim`	Array of at least size `NDim` specifying the result dimensions.
`Nx`	X-dimension of the result.
`Ny`	Y-dimension of the result.
`Orientation`	Applies to 2D results. Specifies the orientation in degrees of the plane wave. This is the wave vector.
`Phase`	Phase of the cosine.

7.46.9 Make an Ellipse or Oblate Spheroid

```
fsignal        mkellipse        (const int NDim,
                                 const int *Dim,
                                 const float *Axes)
fsignal        mkellipse        (const int NDim,
                                 const int *Dim,
                                 const float *Axes,
                                 const int *Center)
fsignal        mkellipse        (const int Nx,
                                 const int Ny,
                                 const float NzOrA,
                                 const float AOrB = -1,
                                 const float B = -1,
                                 const float C = -1)
```

DESCRIPTION:
Create a 2-, or 3D filled ellipse.
This function will fill in an area or volume according to

$$\frac{(x-x_c)^2}{a^2} + \frac{(y-y_c)^2}{b^2} \leq 1$$

for two dimensions; and

$$\frac{(x-x_c)^2}{a^2} + \frac{(y-y_c)^2}{b^2} + \frac{(z-z_c)^2}{c^2} \leq 1$$

for three; where x_c, y_c, z_c represent the pixel center of the ellipse.
The arguments are:

`NDim`	The dimensionality of the result.
`Dim`	Array of at least size `NDim` specifying the result dimensions.
`Axes`	Array of at least size `NDim` specifying the major axes, a, b, and c (see below).
`Center`	Array of at least size `NDim` specifying the location of the center of the ellipse.
`Nx`	X-dimension of the result.
`Ny`	Y-dimension of the result.
`NzOrA`	Z-dimension of the result for three dimensions or the X major axis for two.
`AOrB`	The X major axis for three dimensions or the Y major axis for two.
`B`	The Y major axis for three dimensions.
`C`	The Z major axis for three dimensions.

7.46.10 Make a Complex Wave

```
csignal        mkeuler        (const float Frequency,
                               const float SampleRate,
                               const int NDim,
                               const int *Dim,
                               const float Orientation,
                               const float Phase)
csignal        mkeuler        (const float Frequency,
                               const float SampleRate,
                               const int Nx,
                               const int Ny = 0,
                               const float Orientation = 0,
                               const float Phase = 0)
```

DESCRIPTION:
Create a 1-, or 2D Euler signal.
If f is the frequency in Hertz, and ϕ the phase, then `mkeuler` returns:
$\exp(2\ \pi\ f\ (x\ +\ \phi))$ in 1D, and
$\exp(2\ \pi\ f\ (k_x\ x\ + k_y\ y\ +\ \phi))$ in 2D.
The arguments are:

`Frequency`	Frequency of the complex wave in Hertz.
`SampleRate`	Rate at which the wave should be sampled in Hertz.
`NDim`	The dimensionality of the result.
`Dim`	Array of at least size `NDim` specifying the result dimensions.
`Nx`	X-dimension of the result.
`Ny`	Y-dimension of the result.
`Orientation`	Applies to 2D results. Specifies the orientation in degrees of the plane wave. This is the wave vector.
`Phase`	Phase of the wave.

7.46.11 Make a Function

```
fsignal        mkfield        (const float Slope,
                               const float SampleRate,
                               const int NDim,
                               const int *Dim,
                               const float Orientation,
                               const float Delay,
                               double (*Function)(double))
```

DESCRIPTION:
Create a 1-, or 2D field signal using the function `Function` to generate the values. Let $F(x)$ be the function, s be the slope, and d the delay, then `mkfield` returns

$F(s\ (x\ +\ d))$ in 1D, and
$F(s\ (k_x\ x\ +\ k_y\ y\ +d))$ in 2D.

The arguments are:

`Slope`	Slope, s, of the function. This has units of $1/x$. For example, `Slope` could be units of Hertz if `Function` were `sin`.
`SampleRate`	Rate, r, at which the function should be sampled. This has units of $1/x$ so that the sample interval, h, has units of x. The sample rate is the same for both x and y in two dimensional results.
`NDim`	The dimensionality of the result.
`Dim`	Array of at least size `NDim` specifying the result dimensions.
`Orientation`	Applies to two dimensional results. Specifies the orientation in degrees of the function. This is the angle of the wave vector. θ in the description.
`Delay`	Delay, d, of the function. This is a constant value added to the argument of the function.
`Function`	`double` function taking a single `double` argument to be used to compute the result. $F(x)$ in the description.

7.46.12 Make a Gaussian

```
fsignal        mkgauss        (const double SD,
                               const int NDim,
                               const int *Dim)
fsignal        mkgauss        (const double SD,
                               const int Nx,
                               const int Ny = 0,
                               const int Nz = 0)
```

DESCRIPTION:
Create a 1-, 2-, or 3D Gaussian window.
The arguments are:

`SD`	Standard deviation of the window.
`NDim`	The dimensionality of the result.
`Dim`	Array of at least size `NDim` specifying the result dimensions.
`Nx`	X-size of the window.
`Ny`	Y-size of the window.
`Nz`	Z-size of the window.

7.46.13 Make a Ramp

```
fsignal        mkramp        (const float Slope,
                              const float Intercept,
                              const float SampleRate,
                              const int NDim,
                              const int *Dim,
                              const float Orientation)
fsignal        mkramp        (const float Slope,
                              const float Intercept,
                              const float SampleRate,
                              const int Nx,
                              const int Ny = 0,
                              const float Orientation = 0)
```

DESCRIPTION:

Create a 1-, or 2D ramp signal. If s is the slope, and b the intercept, then `mkramp` returns

$s\,x + b$ in 1D, and

$s\,(k_x\,x\ +\ k_y\,y) + b$ in 2D.

The arguments are:

`Slope`	Slope of the ramp.
`Intercept`	Intercept of the ramp.
`SampleRate`	Rate at which the ramp should be sampled.
`NDim`	The dimensionality of the result.
`Dim`	Array of at least size `NDim` specifying the result dimensions.
`Nx`	X-dimension of the result.
`Ny`	Y-dimension of the result.
`Orientation`	Applies to 2D results. Specifies the orientation in degrees of the ramp.

7.46.14 Make a Sine

```
fsignal        mksin        (const float Frequency,
                             const float SampleRate,
                             const int NDim,
                             const int *Dim,
                             const float Orientation,
                             const float Phase)
fsignal        mksin        (const float Frequency,
                             const float SampleRate,
                             const int Nx,
                             const int Ny = 0,
                             const float Orientation = 0,
                             const float Phase = 0)
```

DESCRIPTION:
Create a 1-, or 2D sine signal.
If f is the frequency in Hertz, and ϕ the phase, then `mksin` returns:
$\sin(2\,\pi\,f\,(x\,+\,\phi))$ in 1D, and
$\sin(2\,\pi\,f\,(k_x\,x\,+k_y\,y\,+\,\phi))$ in 2D.
The arguments are:

`Frequency`	Frequency of the sine in Hertz.
`SampleRate`	Rate at which the sine should be sampled in Hertz.
`NDim`	The dimensionality of the result.
`Dim`	Array of at least size `NDim` specifying the result dimensions.
`Nx`	X-dimension of the result.
`Ny`	Y-dimension of the result.
`Orientation`	Applies to 2D results. Specifies the orientation in degrees of the plane wave. This is the wave vector.
`Phase`	Phase of the sine.

7.46.15 Make a Square Wave

```
fsignal        mksqrwv        (const float Frequency,
                               const float SampleRate,
                               const int NDim,
                               const int *Dim,
                               const float Orientation,
                               const float Phase)
fsignal        mksqrwv        (const float Frequency,
                               const float SampleRate,
                               const int Nx,
                               const int Ny = 0,
                               const float Orientation = 0,
                               const float Phase = 0)
```

DESCRIPTION:
Create a 1-, or 2D square wave signal.
If f is the frequency in Hertz, and ϕ the phase, then `mksqrwv` returns:

$\mathrm{sgn}(\sin(2\,\pi\,f\,(x\,+\,\phi)))$ in 1D, and

$\mathrm{sgn}(\sin(2\,\pi\,f\,(k_x\,x\,+k_y\,y\,+\,\phi)))$ in 2D.

The arguments are:

`Frequency`	Frequency of the square wave in Hertz.
`SampleRate`	Rate at which the square wave should be sampled in Hertz.
`NDim`	The dimensionality of the result.
`Dim`	Array of at least size `NDim` specifying the result dimensions.
`Nx`	X-dimension of the result.
`Ny`	Y-dimension of the result.
`Orientation`	Applies to 2D results. Specifies the orientation in degrees of the plane wave. This is the wave vector.
`Phase`	Phase of the square wave.

7.46.16 Make a Filter Window

```
fsignal        mkwindow        (float *Centers ,
                                float *Widths ,
                                int NumberOfBands ,
                                int Window ,
                                int Reject ,
                                int Size ,
                                float Origin ,
                                float Interval ,
                                int *Start ,
                                int *Stop,
                                int Toggle)
```

DESCRIPTION:
Create a filter window. The filter can be multi-modal with the centers, widths, and number of bands specified in the `Centers`, `Widths`, and `NumberOfBands` arguments.
The arguments are:

`Centers`	An array of size `NumberOfBands` specifying the center of the filter band(s). At least one center must be supplied.
`Widths`	An array of size `NumberOfBands` specifying the widths of the filter band(s). Widths are specified in frequency units and represent the size of the band from end-to-end and not full-width at half-maximum. At least one width must be supplied.
`NumberOfBands`	The number of bands. Must be at least 1.
`Window`	Window specification. Must be one of `HAMMING`, `HANNING`, `BLACKMAN`, `BARTLETT`, `GAUSSIAN`, or `RECTANGULAR`. These constants are described in Global Constants, Section 13.
`Reject`	Set to 1 to make the filter a band-reject instead of band-pass.
`Size`	The number of elements in the filter.
`Origin`	The data origin of the filter. Unless Toggle is 1, the origin is ignored and taken to be zero.
`Interval`	The data sample interval of the filter.
`Start`	The element index of the actual filter start. All elements before element `Start` are zero. `Start` and `Stop` can be used to determine the effective (non-zero or total pass-band) of the filter.
`Stop`	The element index of the actual filter stop. All elements after element `Stop` are zero. `Start` and `Stop` can be used to determine the effective (non-zero or total pass-band) of the filter.
`Toggle`	Set to 1 to specify the construction of a "toggled" or sign-modulated filter. This will put the zero-frequency value at the center of the signal. If set to zero, the origin will be at element zero.

8 Array Class

The Array class is implemented as a template. Throughout this subsection *type* represents the template parameter.

8.1 Constructors

8.1.1 Default Constructor

```
Array<type>          (void)
```

DESCRIPTION:
Default constructor. A zero-length Array is created.

EXAMPLE:

```
Array<int>   Dim ;
Array<float> Origin;
```

8.1.2 Copy Constructor

```
Array<type>          (const Array<type> &Input)
```

DESCRIPTION:
Copy constructor.

8.1.3 Element Constructors

Array<*type*>	(const int Elements)
Array<*type*>	(const *type* *array, const int Elements)
Array<*type*>	(const int Elements, const *type* *array)

DESCRIPTION:
These constructors allocate memory and optionally initialize it with the elements of array which must be of at least size Elements. If array is not specified, the elements of the Array are set to zero.

EXAMPLE:

```
Array<int> Dim( 100 ) ;

int *count = { 1 , 2 , 3 , 4 , 5 } ;
Array<int> A( count , 5 ) ;
```

8.2 Assignment Operators

8.2.1 Array to Array Assignment

Array<*type*>&	operator =	(const Array<*type*> &Input)

DESCRIPTION:
Assignment operator from one Array to another. Array length and data are copied; the old length and data are lost.

EXAMPLE:

```
float x[5]  = { 1 , 2 , 3 , 4 , 5 } ;
float y[10] = {10, 15, 20, 25, 30, 35, 40, 45, 50, 55} ;

// a is a 5 element array initialized to the values in x.
Array<float> a( 5 , x ) ;

// b is a 10 element array initialized to the values in y.
Array<float> b( 10, y ) ;

// a is now an exact copy of b. Its previous length and elements lost.
a = b ;
```

8.2.2 Scalar to Array Assignment

Array<*type*>&	operator =	(const *type* &Scalar)

DESCRIPTION:
Assignment operator from a scalar to an Array. Each element in the Array is assigned the value of Scalar. The Array length is preserved.

EXAMPLE:

```
Array<int> a( 10 ) ;    // a is a 10 element array.
a = 100 ;               // Each element of a now has value 100.
```

8.3 Array Element Access

type&	operator [] (int n)

DESCRIPTION:
Array element access. The operator can be used to set and retrieve an Array element. Range checking is not performed.

EXAMPLE:

```
Array<int> A( 5 ) ;
int x ;
A[ 2 ] = 3 ;        // Set an element
x      = A[ 2 ] ;   // Retrieve an element
```

8.4 Array Casts

type *	(Array<*type*>)

DESCRIPTION:
Cast to pointer, returns the pointer to the Array.

EXAMPLE:

```
Array<float> Z( 100 ) ;
float *Zptr = Z ;
```

8.5 Unary Negation

Array<*type*>	operator -	(const Array<*type*> &a)

DESCRIPTION:
Unary negation. Each element of the Array is negated.

EXAMPLE:

```
const int N = 12 ;
Array<int> a( N ) , b( N );

// a is set somewhere
b = -a ;
```

8.6 Assignment Arithmetic Operators

8.6.1 Array-Array Assignment Arithmetic Operators

Array<*type*>& operator *op*= (const Array<*type*> &Input)

Assignment arithmetic operators from one **Array** variable to another where *op* is one of +, -, *, /. The operators are applied on an element-by-element basis and can be used on **Arrays** of different lengths. The operations cease when the lesser of the two **Array** lengths has been reached.

EXAMPLE:

```
Array<float> a( 10 ) , b( 5 ) ;
a = 2 ;
b = 2 ;

// The first five elements of a are now zero, the last five are still 2.
a -= b ;
```

8.6.2 Scalar-Array Assignment Arithmetic Operators

Array<*type*>& operator *op*= (const *type* &Scalar)

DESCRIPTION:
Assignment arithmetic operators from a scalar to an Array where *op* is one of +, -, *, /. The arithmetic operator is applied to each element of the **Array**.

EXAMPLE:

```
Array<float> a( 10 ) ;
a = 2 ;
a -= 1 ;  // The elements of a are now 1.
```

8.7 Binary Operators

8.7.1 Array-Array Binary Operators

`Array<type>  operator op(const Array<type> &a, const Array<type> &b)`

DESCRIPTION:
Binary arithmetic operators between **Arrays** where *op* is one of +, -, *, /. The operators are applied on an element-by-element basis and can be used on **Arrays** of different lengths. The operations ceases when the lesser of the two **Array** lengths has been reached.

8.7.2 Array-Scalar Binary Operators

`Array<type>  operator op(const Array<type> &a, const type &Scalar)`

DESCRIPTION:
Binary arithmetic operators between **Arrays** and scalars where *op* is one of +, -, *, /.

8.8 Array Relational Operators

```
int          operator ==(const Array<type> &a, const Array<type> &b)
int          operator !=(const Array<type> &a, const Array<type> &b)
```

DESCRIPTION:
Relational operators. Element by element comparison is performed.

8.9 Elements in an Array

```
int          nelemof          (const Array<type> &a)
```

Function to retrieve the number of elements in an `Array`.

EXAMPLE:

```
Array<float> x( 101 ) ;
for( int n=0 ; n<nelemof( x ) ; n++ )
      x[ n ] = 2*n + 1 ;
```

8.10 Stream Operator

```
ostream&     operator <<      (ostream &co, const Array<type> &a)
```

DESCRIPTION:
`ostream` operator to print the contents of an `Array`.

9 Complex Class

The "Complex" class actually consists of three classes:

- `complex`, a `float` complex class;
- `dcomplex`, a `double` complex class;
- `Complex<TYPE>`, a template complex class used to handle our multiple type complex data.

The first two classes are "hard-wired", i.e. not templates. This was needed to support mixed-type computations.
The third class is implemented as a template since we deal with multiple types of complex data. This class supports only arithmetic functions.
Note: All the complex types are defined as

```
private:
    TYPE re , im ;
public:
```

so that when a complex array is stored, the values are saved as

```
re im re im re im ...
```

9.1 Constructors

9.1.1 Default Constructor

`complex`	`(void)`
`dcomplex`	`(void)`
`Complex<`*type*`>`	`(void)`

DESCRIPTION:
Default constructor. Real and imaginary parts of the variable are initialized to zero.

EXAMPLE:

```
complex c ;                        // A complex float.
Complex<unsigned char> IQ[ 256 ] ; // An array of complex unsigned char.
dcomplex z[ 100 ] ;                // An array of complex double.
```

9.1.2 Data Constructor

complex	(const float Real, const float Imag = 0)
dcomplex	(const double Real, const double Imag = 0)
Complex<*type*>	(const *type* Real, const *type* Imag = 0)

DESCRIPTION:
This constructor specifies the real and optionally the imaginary part. If the latter is not specified, it is initialized to zero.

EXAMPLE:

```
dcomplex z( 3.5 ) ;
dcomplex x( 3.5 , 4.1 ) ;
```

9.1.3 Copy Constructor

complex	(const complex &Input)
complex	(const dcomplex &Input)
complex	(const Complex<*type*> &Input)
dcomplex	(const complex &Input)
dcomplex	(const dcomplex &Input)
dcomplex	(const Complex<*type*> &Input)
Complex<*type*>	(const Complex<*type*> &Input)

DESCRIPTION:
Copy constructor.

9.2 Assignment Operators

9.2.1 Complex to Complex Assignment

```
complex&        operator =        (const complex &Input)
complex&        operator =        (const dcomplex &Input)

dcomplex&       operator =        (const complex &Input)
dcomplex&       operator =        (const dcomplex &Input)

Complex<type>&  operator =        (const Complex<type> &Input)
```

DESCRIPTION:
Assignment operator from one `Complex<type>` variable to another.

9.2.2 Scalar to Complex Assignment

```
complex&        operator =        (const float &Scalar)

dcomplex&       operator =        (const double &Scalar)

Complex<type>&  operator =        (const type &Scalar)
```

DESCRIPTION:
Assignment operator from a real scalar to a `Complex<type>` variable. The imaginary part of the variable is set to zero.

EXAMPLE:

```
complex x( 1 , 2 ) ; // x = 1 + i 2
x = 3.14159 ;        // x now equals 3.14159 + i (0)
```

9.3 Assignment Arithmetic Operators

9.3.1 Complex to Complex Assignment Arithmetic Operators

```
complex&        operator op=     (const complex &Input)
complex&        operator op=     (const dcomplex &Input)

dcomplex&       operator op=     (const complex &Input)
dcomplex&       operator op=     (const dcomplex &Input)

Complex<type>&  operator op=     (const Complex<type> &Input)
```

DESCRIPTION:
Assignment arithmetic operators from one complex variable to another where *op* is one of +, -, *, /.

9.3.2 Scalar to Complex Assignment Arithmetic Operators

```
complex&        operator op=     (const float Scalar)

dcomplex&       operator op=     (const double Scalar)

Complex<type>&  operator op=     (const type Scalar)
```

DESCRIPTION:
Assignment arithmetic operators from a real scalar to a complex variable where *op* is one of +, -, *, /.

9.4 Binary Operators

9.4.1 Complex-Complex Binary Operators

```
complex      operator op (const complex &a, const complex &b)

dcomplex     operator op (const dcomplex &a, const dcomplex &b)
dcomplex     operator op (const complex &a, const dcomplex &b)
dcomplex     operator op (const dcomplex &a, const complex &b)

Complex<type> operator op (const Complex<type> &a, const Complex<type> &b)
```

DESCRIPTION:
Binary arithmetic operators between complex variables where *op* is one of +, -, *, /.

9.4.2 Complex-Scalar Binary Operators

```
complex      operator op (const complex &a, const float Scalar)
complex      operator op (const float Scalar, const complex &b)
dcomplex     operator op (const complex &a, const double Scalar)
dcomplex     operator op (const double Scalar, const complex &b)

dcomplex     operator op (const dcomplex &a, const double Scalar)
dcomplex     operator op (const double Scalar, const dcomplex &b)

Complex<type> operator op (const Complex<type> &a, const type Scalar)
```

DESCRIPTION:
Binary arithmetic operators between Complex and real variables where *op* is one of +, -, *, /.

9.5 Extract Real and Imaginary Parts

float&	real	(complex &z)
float&	imag	(complex &z)
double&	real	(dcomplex &z)
double&	imag	(dcomplex &z)
type &	real	(Complex<*type*> &z)
type &	imag	(Complex<*type*> &z)

float	real	(const complex &z)
float	imag	(const complex &z)
double	real	(const dcomplex &z)
double	imag	(const dcomplex &z)
type	real	(const Complex<*type*> &z)
type	imag	(const Complex<*type*> &z)

DESCRIPTION:
The first form can be used to set and retrieve the real and imaginary parts of a complex variable. It can be used on either side of an equation.
The second form operates on `const` variables and can only appear on the right hand side of an equation.
The compiler will determine which function should be used. According to the ARM [1], section 13, for any type `T`, "...It is, however, possible to distinguish between `const T&`, `volatile T&`, and plain `T&` so functions that differ only in this respect may be defined." Thus, in theory, compilers should be able to distinguish between `real(const complex&)` and `real(complex&)`. I, however, have come across some, which under certain circumstances, cannot. The compilers have apparently not read the ARM thoroughly.

EXAMPLE:

```
complex c ;
const complex z( 1 , 2 ) ;
float x ;
real( c ) = 5 ;          // Set the real part
imag( c ) = real( z ) ; // Set the imaginary part
x = real( c ) ;          // Retrieve the real part
```

9.6 Casting

`float`	`(complex)`
`double`	`(dcomplex)`
type	`(Complex<`*type*`>)`

DESCRIPTION:
Cast to real part. This cast to a non-complex type returns the real part of the complex variable.

EXAMPLE:

```
complex z( 3 , 5 ) ;
float x ;
x = z ;  // x is now equal to 3
```

9.7 Unary Negation

`complex`	`operator -`	`(const complex &z)`
`dcomplex`	`operator -`	`(const dcomplex &z)`
`Complex<`*type*`>`	`operator -`	`(const Complex<`*type*`> &z)`

DESCRIPTION:
Unary negation. Real and imaginary parts are negated.

9.8 Relational Operators

```
int     operator ==     (const complex &a, const complex &b)

int     operator ==     (const dcomplex &a, const dcomplex &b)

int     operator ==     (const Complex<type> &a, const Complex<type> &b)
```

```
int     operator !=     (const complex &a, const complex &b)

int     operator !=     (const dcomplex &a, const dcomplex &b)

int     operator !=     (const Complex<type> &a, const Complex<type> &b)
```

```
int     operator >      (const complex &a, const complex &b)

int     operator >      (const dcomplex &a, const dcomplex &b)

int     operator >      (const Complex<type> &a, const Complex<type> &b)
```

```
int     operator <      (const complex &a, const complex &b)

int     operator <      (const dcomplex &a, const dcomplex &b)

int     operator <      (const Complex<type> &a, const Complex<type> &b)
```

DESCRIPTION:
Relational operators.
The equality operator compares real parts and imaginary parts and is equivalent to implementing:

```
(real(a) == real(b)) && (imag(a) == imag(b))
```

The inequality operator compares real parts and imaginary parts and is equivalent to implementing:

```
(real(a) != real(b)) || (imag(a) != imag(b))
```

The greater than and less than operators compare magnitudes, and are equivalent to implementing

```
fabs(a) > fabs(b)
fabs(a) < fabs(b)
```

respectively.

9.9 Math Functions

9.9.1 Phase of a Complex Variable

double	arg	(const complex &z)
double	arg	(const dcomplex &z)
double	arg	(const Complex<*type*> &z)

DESCRIPTION:
Compute and return the argument or phase of a complex variable. If $z = x + iy$ then `arg(z)` returns $\arctan(y/x)$. This function is implemented with `atan2(double,double)`, and returns a `double`, regardless of the input data type.

EXAMPLE:

```
Complex<int> a( 1 , 1 ) ;
double phase = arg( a ) ; // phase is Pi/4.
```

9.9.2 Cube Root

complex	cbrt	(const complex &z)
dcomplex	cbrt	(const dcomplex &z)

DESCRIPTION:
Return the first branch of the cube root of the argument. If $z = x + iy$, then `cbrt(z)` returns:

$$\texttt{cbrt}(z) \quad = \quad \sqrt[6]{x^2 + y^2}\ \exp\left(\arctan(y/x)/3\right)$$

Note: To remain consistent with the `cbrt()` function defined in `math.h`, this should return a `dcomplex` regardless of the input type. However to save time on casting from `dcomplex` to `complex`, the function is overloaded to return the type of its argument. The templated complex types must be explicitly cast to either `dcomplex` or `complex`.

9.9.3 Complex Conjugate

complex	cc	(const complex &z)
dcomplex	cc	(const dcomplex &z)
Complex<*type*>	cc	(const Complex<*type*> &z)

DESCRIPTION:
Return the complex conjugate of the argument. Note: This function is overloaded to accept arguments of non-Complex *type* (`float`, `int`, etc.) so that it may be used in parameterized functions. The complex conjugate of a real variable returns the variable itself.

9.9.4 Cosine

complex	cos	(const complex &z)
dcomplex	cos	(const dcomplex &z)

DESCRIPTION:
Return the cosine of the argument. Note: To remain consistent with the `cos()` function defined in `math.h`, this should return a `dcomplex` regardless of the input type. However to save time on casting from `dcomplex` to `complex`, the function is overloaded to return the type of its argument. The templated complex types must be explicitly cast to either `dcomplex` or `complex`.

9.9.5 Euler Exponential

complex	euler	(const float x)
dcomplex	euler	(const double x)
complex	euler	(const complex z)
dcomplex	euler	(const dcomplex z)

DESCRIPTION:
Return the Euler exponential of the argument. For a real, x, this function returns

$$\begin{aligned} \texttt{euler}(x) &= e^{ix} \\ &= \cos x + i \sin x \end{aligned}$$

For a complex, $z = x + iy$, it returns

$$\begin{aligned} \texttt{euler}(z) &= e^{iz} \\ &= e^{i(x+iy)} \\ &= e^{-y}\, e^{ix} \\ &= e^{-y}\, (\cos x + i \sin x) \end{aligned}$$

Note: To remain consistent with the `euler()` function defined in `math.h`, this should return a `dcomplex` regardless of the input type. However to save time on casting from `dcomplex` to `complex`, the function is overloaded to return the type of its argument. The templated complex types must be explicitly cast to either `dcomplex` or `complex`.

EXAMPLE:
This code sequence shows how the `euler` function can be used to demonstrate the wave function of a particle in the presence of a potential step. Figure 10 shows the wave function and potential barrier.

```
void PotentialStep()
{
     float V(float x) ;
     int
          N = 600,
          n ;
     float
          x0 = -2,
          xf = 1 ,
          dx = (xf - x0)/(float)N ,
          x ,
          E = 0.5 ,      /* Particle energy */
          k = 2*Pi/0.5;

     csignal psi(1,&N,0,&x0,&dx) ;
     fsignal potential(1,&N,0,&x0,&dx) ;

     for( n=0, x=x0 ; n<N ; n++ , x+=dx )
     {
          psi[n]       = euler( k * x * sqrt( complex(E-V(x)) ) ) ;
          potential[n] = V(x) ;
     }
```

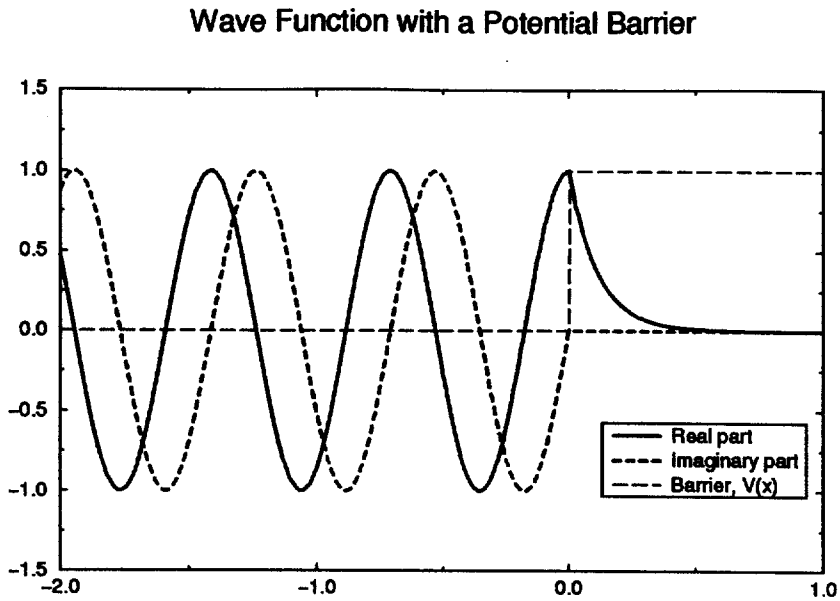

Figure 10: Demonstration of euler function. A particle wave function in the presence of a potential step.

```
    save( psi , "WaveFunction" ) ;
    save( potential , "Barrier" ) ;
}

float V(float x)
{
    float Value ;

    if( x < 0.0 )
        Value = 0.0 ;
    else
        Value = 1.0 ;

    return Value ;
}
```

9.9.6 Exponential

complex	exp	(const complex &z)
dcomplex	exp	(const dcomplex &z)

DESCRIPTION:
Return the exponential of the argument. Note: To remain consistent with the `exp()` function defined in `math.h`, this should return a `dcomplex` regardless of the input type. However to save time on casting from `dcomplex` to `complex`, the function is overloaded to return the type of its argument. The templated complex types must be explicitly cast to either `dcomplex` or `complex`.

9.9.7 Magnitude

double	fabs	(const complex &z)
double	fabs	(const dcomplex &z)
double	fabs	(const Complex<*type*> &z)

DESCRIPTION:
Return the magnitude of the argument. If $z = x + iy$ then `fabs(z)` returns $|z| = \sqrt{x^2 + y^2}$. This function returns a `double`, regardless of the input data type to be consistent with the `fabs()` function defined in `math.h`.

9.9.8 Intensity

double	intensity	(const complex &z)
double	intensity	(const dcomplex &z)
double	intensity	(const Complex<*type*> &z)

DESCRIPTION:
Return the magnitude of the argument. If $z = x+iy$ then `intensity(z)` returns $|z|^2 = x^2+y^2$. This function returns a `double`, regardless of the input data type to be consistent with the `fabs()` function defined in `math.h`.

9.9.9 Natural Logarithm

complex	log	(const complex &z)
dcomplex	log	(const dcomplex &z)

DESCRIPTION:
Return the natural logarithm of the argument. Note: To remain consistent with the `log()` function defined in `math.h`, this should return a `dcomplex` regardless of the input type. However to save time on casting from `dcomplex` to `complex`, the function is overloaded to return the type of its argument. The templated complex types must be explicitly cast to either `dcomplex` or `complex`.

9.9.10 Base 10 Logarithm

complex	log10	(const complex &z)
dcomplex	log10	(const dcomplex &z)

DESCRIPTION:
Return the base 10 logarithm of the argument. Note: To remain consistent with the `log10()` function defined in `math.h`, this should return a `dcomplex` regardless of the input type. However to save time on casting from `dcomplex` to `complex`, the function is overloaded to return the type of its argument. The templated complex types must be explicitly cast to either `dcomplex` or `complex`.

9.9.11 Convert from Magnitude and Phase to Real and Imaginary

`complex`	`mptori`	`(const float &mag, const float &phase)`
`dcomplex`	`mptori`	`(const double &mag, const double &phase)`
`Complex<`*type*`>`	`mptori`	`(const` *type* `&mag, const` *type* `&phase)`

DESCRIPTION:
Convert two real numbers representing the magnitude and phase of a complex number to a complex number in real and imaginary format. If A is the magnitude and θ is the phase, then `mptori` returns the following complex number:

$$\begin{aligned} z &= Ae^{i\theta} \\ &= A\cos(\theta) + iA\sin(\theta) \end{aligned}$$

9.9.12 Convert to Polar Form

`complex`	`polar`	`(const complex &z)`
`dcomplex`	`polar`	`(const dcomplex &z)`
`Complex<`*type*`>`	`polar`	`(const Complex<`*type*`> &z)`

DESCRIPTION:
Compute and return the polar form of the input. Assumes the argument is in rectangular format. If $z = x+iy$ then `polar(`z`)` returns $\sqrt{x^2+y^2}\, e^{i\arctan(y/x)}$, with the real part equal to $\sqrt{x^2+y^2}$, and the imaginary part to $\arctan(y/x)$.

9.9.13 Power

complex	pow	(const complex &z, const double p)
dcomplex	pow	(const dcomplex &z, const double p)

DESCRIPTION:
Return the argument raised to the power p.

9.9.14 Convert to Rectangular Form

complex	rect	(const complex &z)
dcomplex	rect	(const dcomplex &z)
Complex<*type*>	rect	(const Complex<*type*> &z)

DESCRIPTION:
Compute and return the number in rectangular format. Assumes the argument is in polar format. If $z = Ae^{i\theta}$ (stored as (A, θ)), then `rect(z)` returns $A\cos(\theta) + iA\sin(\theta)$.

9.9.15 Return Rectangular Parts

```
double      rect_x      (const complex &z)
double      rect_y      (const complex &z)

double      rect_x      (const dcomplex &z)
double      rect_y      (const dcomplex &z)

double      rect_x      (const Complex<type> &z)
double      rect_y      (const Complex<type> &z)
```

DESCRIPTION:
Compute and return the X- or Y-coordinate of a number in polar format. Assumes the argument is in polar format, i.e. is stored as (A,θ) where $z = Ae^{i\theta}$. Then `rect_x(z)` returns $A\cos(\theta)$, and `rect_y(z)` returns $A\sin(\theta)$.

9.9.16 Sine

```
complex     sin         (const complex &z)

dcomplex    sin         (const dcomplex &z)
```

DESCRIPTION:
Return the sine of the argument. Note: To remain consistent with the `sin()` function defined in `math.h`, this should return a `dcomplex` regardless of the input type. However to save time on casting from `dcomplex` to `complex`, the function is overloaded to return the type of its argument. The templated complex types must be explicitly cast to either `dcomplex` or `complex`.

9.9.17 Square Root

complex	sqrt	(const complex &z)
dcomplex	sqrt	(const dcomplex &z)

DESCRIPTION:
Return the first branch of the square root of the argument. If $z = x + iy$, then `sqrt(z)` returns:

$$\texttt{sqrt}(z) \quad = \quad \sqrt[4]{x^2 + y^2}\ \exp\left(\arctan(y/x)/2\right)$$

Note: To remain consistent with the `sqrt()` function defined in `math.h`, this should return a `dcomplex` regardless of the input type. However to save time on casting from `dcomplex` to `complex`, the function is overloaded to return the type of its argument. The templated complex types must be explicitly cast to either `dcomplex` or `complex`.

9.10 Stream Operators

ostream &	operator <<	(ostream &co, const complex &z)
ostream &	operator <<	(ostream &co, const dcomplex &z)
ostream &	operator <<	(ostream &co, const Complex<*type*> &z)

DESCRIPTION:
`ostream` operator to print the contents of a complex variable.

9.11 Show

```
void        show        (const char *Message,
                         const complex &a,
                         FILE *fp = stdout)

void        show        (const char *Message,
                         const dcomplex &a,
                         FILE *fp = stdout)

void        show        (const char *Message,
                         const Complex<type> &a,
                         FILE *fp = stdout)
```

DESCRIPTION:
Function to show a complex variable in a fancy manner.

EXAMPLE:
The code

```
const complex a( 10 , .12 ) ;
show( "a" , a , stdout) ;
```

will produce:

```
a = { 10 , .12 }
```

9.12 Complex Class Type Definitions

For convenience, the following complex type definitions exist:

```
typedef Complex<char> & ccomplex
typedef Complex<short> & scomplex
typedef Complex<unsigned char> & uccomplex
typedef Complex<int> & icomplex
```

For example:

```
uccomplex  z ;        // z is Complex<unsigned char>
scomplex  a[ 10 ] ;   // A Complex<short> array.
```

10 String Class

A `String` is a variable used to store and manipulate character strings.

10.1 Constructors

10.1.1 Default Constructor

`String`	`(void)`

DESCRIPTION:
General constructor requiring no arguments. A zero-length `String` is created.

EXAMPLE:

```
String Name ;
```

10.1.2 Copy Constructor

`String`	`(const String &Input)`

DESCRIPTION:
Copy constructor.

10.1.3 Constructor from a Character Array

`String`	`(const char *CharacterString)`

DESCRIPTION:
Constructor from a pointer to `char`.

EXAMPLE:

```
String Name( "Sean K. Lehman" ) ;
```

10.1.4 Constructor from a Letter

`String`	`(const char &Letter)`

DESCRIPTION:
Constructor initializing the `String` by a `char`, i.e. a single character.

EXAMPLE:

```
String a( 'a' ) ;
String b( 'b' ) ;
```

10.1.5 Constructor from an Integer

`String`	`(const int &Integer)`

DESCRIPTION:
Constructor initializing the `String` by an `int`.

EXAMPLE:

```
String three( 3 ) ;
```

10.2 Casts

`char*`	`(String)`
`int`	`(String)`

DESCRIPTION:
The first cast returns the pointer to the `String`'s `char` array.
The second cast returns the size of the `String`.

EXAMPLE:

```
String Name( "Sean K. Lehman" ) ;
fprintf( stdout , "My name is %s\n" , (char*)Name ) ;

fprintf( stdout , "I have %d letters in my name.\n" , (int)Name ) ;
```

10.3 Assignment Operators

String&	operator =	(const String &Input)

DESCRIPTION:
Assignment operator from one `String` to another.

10.4 Assignment Arithmetic Operators

10.4.1 Assignment Addition

String&	operator +=	(const String &Input)

DESCRIPTION:
Assignment increment between `Strings`. The right hand `String` is concatenated to the left hand `String`.

EXAMPLE:

```
String Basename( "radar" ) ;
Basename += ".sdt" ; // Basename is "radar.sdt"
```

10.4.2 Assignment Subtraction

String&	operator -=	(const String &Input)

DESCRIPTION:
Assignment decrement between `Strings`. The first occurrence of the right hand `String` is deleted from the left hand `String`.

EXAMPLE:

```
String Basename( "radar.sdt" ) ;
Basename -= ".sdt" ; // Basename is "radar"
```

10.5 Binary Operators

10.5.1 String-String Binary Operators

String	operator +	(const String &a, const String &b)
String	operator -	(const String &a, const String &b)

DESCRIPTION:
Binary addition and subtraction between Strings. Addition concatenates the Strings, subtraction deletes the first occurrence of the second String from the first.

10.5.2 Char-String Binary Operators

String	operator +	(const char *a, const String &b)
String	operator -	(const char *a, const String &b)

DESCRIPTION:
Binary addition and subtraction between Strings and char arrays. Addition concatenates the arguments, subtraction deletes the first occurrence of the String from the char array.

10.5.3 String-Char Binary Operators

String	operator +	(const String &a, const char *b)
String	operator -	(const String &a, const char *b)

DESCRIPTION:
Binary addition and subtraction between Strings and char arrays. Addition concatenates the arguments, subtraction deletes the first occurrence of the char array from the String.

10.6 Relational Operators

```
int          operator ==      (const String &a,const String &b)
int          operator !=      (const String &a,const String &b)
```

DESCRIPTION:
Relational operators. These test for equality and inequality of `String` contents and length.

10.7 Show String Contents

```
void         show             (const char *Message,
                               const String &a,
                               FILE *fp = stdout)
void         showflat         (const char *Message,
                               const String &a,
                               FILE *fp = stdout)
```

DESCRIPTION:
Functions to show a `String` in a fancy manner.
The first form puts carriage returns after the braces, the second form does not.

EXAMPLE:

```
String Name( "Sean K. Lehman" ) ;

show( "My name" , Name ) ;
fprintf(stdout,"\n") ;
showflat( "My name" , Name ) ;
```

would produce

```
My name = {
     14 , 0x3c9d0 "Sean K. Lehman"
     }

My name = { 14 , 0x3c9d0 "Sean K. Lehman" }
```

10.8 Parse a UNIX Path

10.8.1 Get UNIX Path Base

```
String      getbase      (const String &Path)
String      getbase      (const char *Path)
```

DESCRIPTION:
These functions return the base name of the argument, assumed to be a UNIX path. The base name is everything after the last directory in the path. The following table shows how various paths are parsed into directory and base names:

PATH	DIRECTORY	BASE NAME
/	/	/
./	.	.
../	.	..
..	.	..
/roi3	/	roi3
/roi3/	/	roi3
/roi3/lehman	/roi3	lehman
/roi3/lehman/	/roi3	lehman
/roi3/lehman/foo	/roi3/lehman	foo
../foo	..	foo
../foo/	..	foo
../foo/bar	../foo	bar
bar	.	bar

10.8.2 Get UNIX Path Directory

```
String      getdir       (const String &Path)
String      getdir       (const char *Path)
```

DESCRIPTION:
These functions return the directory path of the argument, assumed to be a UNIX path. The directory name is everything up to but not including the last /. The table listed in above shows how various paths are parsed into directory and base names.

11 View I/O Support Functions

The View I/O package provides support for reading, writing, and getting parameters from View signals. To use the package include `viewio.h` in your sources and link with `viewio.o` and `libm.a`. All functions listed here require only two files and not the full IDP++ archive.

11.1 Append Text to SPR File

`int AppendToSPRFile (const char *ViewFileName, const char *Text)`

DESCRIPTION:
Append a text string to the end of a View parameter (`.spr`) file. `Text` points to the null-terminated string to be appended.
The function returns `SUCCESS` upon successful completion, and `FAILURE` otherwise. These constants are described in Section 13.
The arguments are:

`ViewFileName`	View signal file base name, do not include "`.sdt`" or "`.spr`."
`Text`	Null-terminated string to be appended.

EXAMPLE:
This code sequence shows how to save the command line arguments to an existing `.spr` file:

```
main(int argn,char *args[])
{
     /*
      * Save command line.
      */
     String Comments = args[0] ;
     for( int n=1 ; n<argn ; n++ )
          Comments += String(" ") + String(args[n]) ;

     AppendToSPRFile( "foo" , Comments ) ;
}
```

11.2 Remove View Signal from Disk

```
int     DeleteViewSignal        (const char *ViewSignalName)
```

DESCRIPTION:
Delete View signal from disk. This function will remove both the ".sdt" and ".spr." files.
The function returns SUCCESS upon successful completion, and FAILURE otherwise. These constants are described in Section 13.

11.3 Retrieve Signal Dimension from the Parameter File

```
int             dimof           (const char *ViewFileName, const int Axis)
```

DESCRIPTION:
Return the dimension of the specified axis of a View signal stored on disk. ViewFileName is a View signal file base name, do not include ".sdt" or ".spr." Axis is one of the axis specifiers (X_AXIS, Y_AXIS, Z_AXIS, T_AXIS) described in the subsection on Global Constants, Section 13.

11.4 Retrieve Signal Dimensionality from the Parameter File

```
int             ndimof          (const char *ViewFileName)
```

DESCRIPTION:
Return the dimensionality of a View signal stored on disk. ViewFileName is a View signal file base name, do not include ".sdt" or ".spr." This routine reads the .spr file.

11.5 Retrieve Signal Interval from the Parameter File

```
float          intervalof   (const char *ViewFileName, const int Axis)
```

DESCRIPTION:
Return the sample interval of the specified axis of a View signal stored on disk. `ViewFileName` is a View signal file base name, do not include "`.sdt`" or "`.spr`." `Axis` is one of the axis specifiers (`X_AXIS`, `Y_AXIS`, `Z_AXIS`, `T_AXIS`) described in the subsection on Global Constants, Section 13.

11.6 Determine If A File Is An IDP++ File

```
int            isidp            (const char *Filename)
```

DESCRIPTION:
Determine if `Filename` is a base name for an IDP++ signal. This function will return 1 if both the `.sdt` and `.spr` files exist, and 0 otherwise.

11.7 Open a View File for Reading

```
FILE*          OpenView          (const char *ViewFileName,
                                  int *Type,
                                  int *NDim,
                                  int *Dim,
                                  float *Origin,
                                  float *Interval
                                  char *Comments = 0)
```

DESCRIPTION:
Open a View data (".sdt") file and return a stream pointer to the file. The View signal parameters are returned in the function's argument list.
The file can be close with `fclose(FILE*)`.
The function returns a `FILE*` upon a successful opening and reading of the parameter file, and opening of the data file. Otherwise a `(FILE*)0` is returned.
The arguments are:

`ViewFileName`	View signal file name stored on disks. Do not include ".sdt" or ".sdt."
`Type`	The signal data type. This will return one of View's recognized data types: `BYTE`, `WORD`, `LWORD`, `REAL`, or `COMPLEX`. These variables are described in the subsection on Global Constants, Section 13.
`NDim`	The dimensionality of the signal.
`Dim`	An array of at least size `MAXDIM`.
`Origin`	An array at least of size `MAXDIM`.
`Interval`	An array of at least size `MAXDIM`.
`Comments`	If not zero, the comments at the end of the `.spr` file will be returned.

11.8 Retrieve Signal Origin from the Parameter File

```
float          originof      (const char*ViewFileName, const int Axis)
```

DESCRIPTION:
Return the origin of the specified axis of a View signal stored on disk. `ViewFileName` is a View signal file base name, do not include ".sdt" or ".spr." `Axis` is one of the axis specifiers (`X_AXIS`, `Y_AXIS`, `Z_AXIS`, `T_AXIS`) described in the subsection on Global Constants, Section 13.

11.9 Read Parameters from .spr File

```
int            ReadSPRFile       (const char *ViewFileName,
                                  const int *Type,
                                  const int *NDim,
                                  const int *Dim,
                                  const float *Origin,
                                  const float *Interval,
                                  const char *Comments = 0)
```

DESCRIPTION:
Read the information from a parameter (`.spr`) file.
The function returns `SUCCESS` upon successful completion, and `FAILURE` otherwise. These constants are described in Section 13.
The arguments are:

`ViewFileName`	View signal file name stored on disk. Do not include ".sdt" or ".spr."
`Type`	The signal data type. This will return one of View's recognized data types: `BYTE`, `WORD`, `LWORD`, `REAL`, or `COMPLEX`. These variables are described in the subsection on Global Constants, Section 13.
`NDim`	The dimensionality of the signal.
`Dim`	An array of at least size `MAXDIM`.
`Origin`	An array at least of size `MAXDIM`.
`Interval`	An array of at least size `MAXDIM`.
`Comments`	If not zero, the comments at the end of the `.spr` file will be returned.

11.10 Read View Data File

There are two classes of functions to read a View data file. The first reads data of any of the types listed in Section 13.1 and converts them to one of IDP++'s four types. The second form is a template and reads the types without any conversion.

11.10.1 Read Data As One of IDP++'s Types

```
int          TypeReadView     (FILE *fp,
                               const int Type ,
                               const int Elements ,
                               type *Data )

int          TypeReadView     (FILE *fp,
                               const int Type,
                               const int NDim,
                               const int *Dim,
                               const int *Start,
                               const int *Size,
                               type *Data)
```

DESCRIPTION:
Read data from an open View file and convert them to one of IDP++'s four types (`float`, `double`, `complex`, `dcomplex`.
Complex data types read in as real are converted to magnitude. Real data types are read in as complex are converted to complex with zero imaginary part.
The second form of the function performs a partial read of the file. The start and size of the read are specified in the `Start` and `Size` arrays, respectively.
Open the file with `OpenView` (Section 11.7). This function returns the number of elements read which is hopefully `Elements` (unless there is an error or end of file).
The arguments are:

`fp`	File pointer as returned by `OpenView`.
`Type`	The signal data type as returned by `OpenView`.
`NDim`	This signal dimensionality as returned by `OpenView`.
`Dim`	The signal dimensions as returned by `OpenView`.
`Elements`	The number of elements to read.
`Start`	The starting location for partial reads.
`Size`	The size of the partial read.
`Data`	Pointer to memory of at least size `Elements*sizeof(`*stype*`)`

11.10.2 Read Data without Conversion

`int`	`ReadView`	`(FILE *fp, int Elements,` *type* `*Data)`

DESCRIPTION:
Read data from an open View file. This function is implemented as a template and, thus, will read any data type. The template parameter is *TYPE*.
Open the file with `OpenView` (Section 11.7). This function returns the number of elements read which is hopefully `Elements` (unless there is an error or end of file).
The arguments are:

`fp`	File pointer as returned by `OpenView`.
`Elements`	The number of elements to read.
`Data`	Pointer to memory of at least size `Elements*sizeof(`*type*`)`.

11.11 Read Rows and Columns of 2D View Files

These functions read rows and columns of data from View files. They operate only on 2D data sets. They are implemented as templates an thus are able to read any data type.

11.11.1 Read a Column

```
int          ReadViewColumn    (const char *ViewFileName,
                               type *Data ,
                               int Column ,
                               int RowStart = 0 ,
                               int Rows = -1)

int          ReadViewColumn    (FILE *fp,
                               type *Data ,
                               int *Dim ,
                               int Column = -1 ,
                               int RowStart = 0 ,
                               int Rows = -1)
```

DESCRIPTION:
Read a column of data from a file. The memory must already be allocated. These functions are implemented as templates and, thus, will read any data type. The template parameter is *type*.
The first form will open the file, read a single column, and close the file.
The second form will read a column of data from an open file pointer. This can be used to read successive columns from an open data file. Open the file with `OpenView` (Section 11.7).
The function returns `SUCCESS` upon successful completion, and `FAILURE` otherwise. These constants are described in Section 13.
The arguments are:

`ViewFileName`	View signal file name stored on disk. Do not include "`.sdt`" or "`.spr`."
`fp`	File pointer opened with `OpenView`.
`Data`	Pointer to existing memory.
`Dim`	Pointer to an array of at least size 2 containing the dimensions of data.
`Column`	Column number to read. In the second form, this defaults to the current column which, the first time through, is the first column.
`RowStart`	Starting row to read. Defaults to first row.
`Rows`	Number of rows to read. Defaults to all rows.

11.11.2 Read a Row

```
int          ReadViewRow      (const char *ViewFileName,
                               type *Data ,
                               int Row ,
                               int ColumnStart = 0 ,
                               int Columns = -1)

int          ReadViewRow      (FILE *fp,
                               type *Data ,
                               int *Dim ,
                               int Row = -1 ,
                               int ColumnStart = 0 ,
                               int Columns = -1)
```

DESCRIPTION:
Read a row of data from a file. The memory must already be allocated. These functions are implemented as templates and, thus, will read any data type. The template parameter is *type*.
The first form will open the file, read a single row, and close the file.
The second form will read a row of data from an open file pointer. This can be used to read successive rows from an open data file. Open the file with `OpenView` (Section 11.7).
The function returns `SUCCESS` upon successful completion, and `FAILURE` otherwise. These constants are described in Section 13.
The arguments are:

`ViewFileName`	View signal file name stored on disk. Do not include ".sdt" or ".spr."
`fp`	File pointer opened with `OpenView`.
`Data`	Pointer to existing memory.
`Dim`	Pointer to an array of at least size 2 containing the dimensions of data.
`Row`	Row number to read. In the second form, this defaults to the current row which, the first time through, is the first row.
`ColumnStart`	Starting column to read. Defaults to first column.
`Columns`	Number of columns to read. Defaults to all columns.

11.12 Resolve a File Name to a View File Name

```
int          ResolveName      (const char *ViewFileName,
                               const char *Extension,
                               String &Fullname)
```

DESCRIPTION:
Resolve a View file name with a partial extension. The `String`, `Fullname`, will be returned with the file name without ".sdt" or ".spr." if successful.
`SUCCESS` is returned upon successful resolution of the name into a View file name. `FAILURE` is returned otherwise.
The arguments are:

`ViewFileName`	Partial View signal file name expanded up to the base name of the data (`.sdt`) or parameter file (`.spr`).
`Extension`	Either ".sdt" or ".spr".
`Fullname`	`String` containing the View name.

EXAMPLE:
If you pass "`foobar.s`" and ".spr", `ResolveName` will return "`foobar`" in `Fullname` if the file "`foobar.spr`" exists.

11.13 Saving

```
int          SaveAsView      (const char *ViewFileName,
                              const int Type,
                              const int NDim,
                              const int *Dim,
                              const type *Data,
                              const float*Origin = 0,
                              const float*Interval = 0,
                              const char *Comments = 0)
```

DESCRIPTION:
Save a data set as a View file.
This function is implemented as a template and, thus, will read any data type. The template parameter is *type*.
The function returns SUCCESS upon successful completion, and FAILURE otherwise. These constants are described in Section 13.
The arguments are:

ViewFileName	View signal file name stored on disk. Do not include ".sdt" or ".spr."
Type	The signal data type. This must be one of IDP++'s recognized data types: CHAR, SHORT, LONG, FLOAT, COMPLEX, INT, DOUBLE, UCHAR, DCOMPLEX, or UCCOMPLEX. These variables are described in the subsection on Global Constants, Section 13.
NDim	The dimensionality of the signal.
Dim	An array of at least size NDim specifying the signal's dimensions.
Data	Data pointer.
Origin	An array of at least size NDim specifying the signal's axis origins.
Interval	An array of at least size NDim specifying the signal's axis intervals.
Comments	Optional comments to be appended at the end of the .spr file.

11.14 Read Signal Type from a Parameter File

int	stypeof	(const char *ViewFileName)

DESCRIPTION:
Return the data type of a View signal stored on disk. This will return one of the recognized signal types described in the subsection on Global Constants, Section 13.
`ViewFileName` is a View signal file base name, do not include "`.sdt`" or "`.spr`." This routine reads the `.spr` file.

EXAMPLE: The following code segment ensure the correct `demean()` function is called based upon the data type of the input file name (in `args[1]`):

```
main(int argn,char *args[])
{
     char *Input = args[ 1 ] ;
     String Name( Input ) ;
     Name += "_dm" ;

     switch( stypeof( Input ) )
     {
     case CHAR:
     case UCHAR:
     case SHORT:
     case LONG:
     case INT:
     case FLOAT:
          save( demean( fsignal(Input) ) , Name ) ;
          break ;
     case UCCOMPLEX:
     case COMPLEX:
          save( demean( csignal(Input) ) , Name ) ;
          break ;
     case DOUBLE:
          save( demean( dsignal(Input) ) , Name ) ;
          break ;
     case DCOMPLEX:
          save( demean( dcsignal(Input) ) , Name ) ;
          break ;
     }
}
```

11.15 Write Parameters

```
int          WriteSPRFile      (const char *ViewFileName,
                                const int Type,
                                const int NDim,
                                const int *Dim,
                                const float *Origin = 0,
                                const float *Interval = 0,
                                const char *Comments = 0)
```

DESCRIPTION:
Create a `.spr` file. This create a new one if it does not exist, and over-write an existing file.
The function returns `SUCCESS` upon successful completion, and `FAILURE` otherwise. These constants are described in Section 13.
The arguments are:

`ViewFileName`	View signal file name stored on disk. Do not include ".`spr`."
`Type`	The signal data type. This must be one of the recognized data types described in the subsection on Global Constants, Section 13.
`NDim`	The dimensionality of the signal.
`Dim`	An array of at least size `NDim` specifying the signal's dimensions.
`Origin`	Optional array of at least size `NDim` specifying the signal's axis origins. If not specified the origins are set to zero.
`Interval`	Optional an array of at least size `NDim` specifying the signal's axis intervals. If not specified the intervals are set to 1.
`Comments`	Optional comments to be appended at the end of the `.spr` file.

12 Support Functions

These are lower level functions which operate on scalars or arrays but not on any of the higher classes. Some of these functions are implemented as templates. Throughout this subsection *type* represents the template parameter.

12.1 Get/Set Points Within Memory

type	getpt	(const *type* *Input, const int i, const int j, const int Nx)
type&	getpt	(*type* *Input, const int i, const int j, const int Nx)
type	getpt	(const *type* *Input, const int i, const int j, const int k, const int Nx, const int Ny)
type&	getpt	(*type* *Input, const int i, const int j, const int k, const int Nx, const int Ny)
type	getpt	(const *type* *Input, const int i, const int j, const int k, const int l, const int Nx, const int Ny, const int Nz)
type&	getpt	(*type* *Input, const int i, const int j, const int k, const int l, const int Nx, const int Ny, const int Nz)

DESCRIPTION:
These functions will retrieve and set points within 2, 3, or 4 dimensional data sets stored in memory. They are implemented as `inline` template functions and should operate fairly fast.
Note: Some compilers may object to the overloading of

```
getpt(const TYPE&, ...)
```

and

```
getpt(TYPE&, ...)
```

claiming they cannot distinguish between the two. They are wrong. According to the ARM [1] Chapter 13, for any type `T`, "... it is possible to distinguish between `const T&`, `volatile T&`, and plain `T&` so functions that differ only in this respect may be defined."
The arguments are:

Input	Pointer to memory.
i	X-coordinate.
j	Y-coordinate.
k	Z-coordinate.
l	t-coordinate.
Nx	X-dimension of data set.
Ny	Y-dimension of data set.
Nz	Z-dimension of data set.

12.2 Maximum/Minimum of Two Numbers

type	MAX	(const *type* a, const *type* b)
type	MIN	(const *type* a, const *type* b)

DESCRIPTION:
Functions to return the maximum/minimum of two scalars. They are intended to replace the preprocessor macros:

```
#define MAX( a , b )  ((a)>(b)?(a):(b))
#define MIN( a , b )  ((a)<(b)?(a):(b))
```

which all C/C++ books give as an example of bad programming.
They are declared as `inline` templates and should therefore operate on any type very fast.

12.3 Maximum/Minimum of an Array

type	MAX	(const *type* *x, const int N)
type	MIN	(const *type* *x, const int N)

DESCRIPTION:
Functions to return the maximum/minimum of an array of size `N`.
They are declared as `inline` templates and should therefore operate on any type very fast.

12.4 Median of an Array

type	median	(const *type* *x,const int N)

DESCRIPTION:
Function to return the median of an array of size `N`. The median of a set of N values is:

$$\mathrm{med}\,(x_i) = \begin{cases} x_{(\nu)} & N = 2\nu + 1 \\ \frac{1}{2}\left(x_{(\nu-1)} + x_{(\nu)}\right) & N = 2\nu \end{cases}$$

The median selects the middle value or average of the two middle values if the number of points is even.
This functions is declared as an `inline` template and should therefore operate on any type very fast.

12.5 Sort an Array

```
void        shsort        (type *x, int N)
```

DESCRIPTION:
Shell sort an array of `N` elements, arranging the elements from lowest in value to highest. This performs an in-place sort.

12.6 Uniformly Distributed Random Variable

```
float       ran1          (int *seed)
float       uniformrandom (const float mean,
                           const float range,
                           int *seed)
```

DESCRIPTION:
Return a random number uniformly distributed between [0,1]. Call with `seed` a negative integer to initialize; thereafter, do not alter it between successive deviates in a sequence.

12.7 Gamma Distributed Random Variable

```
float       gammarandom   (const float alpha,
                           const float beta,
                           int *seed)
```

12.8 Gaussian Distributed Random Variable

```
double      gaussianrandom    (int *seed)
float       gaussianrandom    (const float mean,
                               const float sd,
                               int *seed)
```

12.9 Single Point Spline Interpolation

```
float       evaluate          (const float xvalue,
                               const int N
                               const float *x
                               const float *y)

double      evaluate          (const double xvalue,
                               const int N
                               const double *x
                               const double *y)
```

DESCRIPTION:
Given a list of x and y values, perform a one-time single point spline evaluation at the point `xvalue`. See Section 7.43 for the full set of interpolation functions.

`xvalue`	X-value where interpolation is performed.
`N`	Size of data arrays
`x`	Pointer to the existing X-values. These are the knots of the spline.
`y`	Pointer to the existing Y-values. These are the values at the knots.

13 Global Constants

13.1 Data Type Constants

CONSTANT	VALUE	COMMENT
CHAR	0	char. Equivalent to View's BYTE.
SHORT	1	short. Equivalent to View's WORD.
LONG	2	long. Equivalent to View's LWORD.
FLOAT	3	float. Equivalent to View's REAL.
COMPLEX	4	complex. Floating point complex. Equivalent to View's COMPLEX.
INT	5	int.
DOUBLE	6	double.
UCHAR	7	unsigned char. unsigned char.
DCOMPLEX	8	dcomplex. Double complex.
UCCOMPLEX	9	uccomplex. unsigned char complex.
MIN_DATA_TYPE	CHAR	Minimum data type.
MAX_DATA_TYPE	UCCOMPLEX	Maximum data type.

13.2 Axis Constants

CONSTANT	VALUE	COMMENT
MAXDIM	4	Largest supported dimension. IDP++ supports up to 4 dimensions: X, Y, Z, and time. Note this is just nomenclature, any of the axes of a data set could equally well represent wavelength, temperature, velocity, etc.
X_AXIS	0	Symbolic representation of X-axis.
Y_AXIS	1	Symbolic representation of Y-axis.
Z_AXIS	2	Symbolic representation of Z-axis.
T_AXIS	3	Symbolic representation of t-axis.

13.3 Constants Used in Various Functions

Constants used to set the functionality of the FFT functions. The functionality is controlled by the logical OR of these constants. For example,

```
fsignal Input ;
csignal Result ;
   ...
Result = fft( Input , FFT_TB | FFT_SP | FFT_HP ) ;
          ...
```

would result in a half-plane, positive sign transform with the input data set being toggled before the FFT.

CONSTANT	VALUE	COMMENT
FFT_TB	0x01	Toggle before.
FFT_TA	0x02	Toggle after.
FFT_HP	0x04	Set half plane transform.
FFT_NM	0X08	Normalize result.
FFT_SP	0x10	Set transform sign positive.
FFT_SN	0x20	Set transform sign negative.

Constants used in the `normalize` function to specify the type of normalization to be performed.

CONSTANT	VALUE	COMMENT
NORM_TO_MAX	0	Normalize to maximum signal value.
NORM_TO_SUM	1	Normalize to signal sum.
NORM_TO_VAR	2	Normalize signal to have unit variance.
NORM_TO_SUMFABS	3	Normalize to sum of the magnitudes.
NORM_TO_MAXMAG	4	Normalize to maximum magnitude.
NORM_TO_MEAN	5	Normalize signal to have unit mean.

Constants used in the `resample` and `magnify` functions to indicate the type of interpolation to be performed.

CONSTANT	VALUE	COMMENT
R_BILINEAR	0	Selects bilinear interpolation.
R_SPLINE	1	Selects spline interpolation.
R_REPLICATE	2	Selects pixel replication.
R_ZEROINS	3	Selects zero insertion.

Constants used in the `mkwindow` function to indicate the window type.

CONSTANT	VALUE	COMMENT
HAMMING	0	Selects a Hamming window.
HANNING	1	Selects a Hanning window.
BLACKMAN	2	Selects a Blackman window.
BARTLETT	3	Selects a Bartlett window.
GAUSSIAN	4	Selects a Gaussian window.
RECTANGULAR	5	Selects a rectangular window.

Constants used in the `ascii` function to select whether signal coordinates should be written with the data.

CONSTANT	VALUE	COMMENT
WRA_NONE	0	No coordinates or indices will be written.
WRA_INDEX	1	Indices will be written.
WRA_COOR	2	Coordinates will be written.

13.4 Miscellaneous Constants

CONSTANT	VALUE	COMMENT
SUCCESS	0	Indication of successful return from a function.
FAILURE	1	Indication of unsuccessful return from a function.
OTHER	-1	Another choice for a return status from a function.
Pi	3.14159...	`4. * atan( 1. )`
GoldenRatio	1.61803...	`(1 + sqrt( 5. ))/2.`

A What's New Since Version 3.5?

The signal and complex classes were overhauled eliminating their dependence upon templates which do not appear to be implemented in a robust manner. Templates are a great idea but they do not solve all the world's problems, even when compilers implement them properly. There appear to be two basic problems with templates:

1. The inability of instantiations of differing types to communicate with each other (e.g. adding a `Complex<float>` to a `Complex<int>` assigning to a `Complex<double>`;
2. Compilers are just not up to supporting heavy template usage yet (although the situation is slowly getting better).

Hence, the lofty goal of supporting ten signal types implemented via templates has been reduced supporting to four "hard wired" classes:

`fsignal`:	A 4-byte floating point data type (float).
`dsignal`:	An 8-byte floating point data type (double).
`csignal`:	An 8-byte floating point data type with real and imaginary parts (complex float).
`dcsignal`:	A 16-byte floating point data type with real and imaginary parts (complex double).

All of the ten previous signal data types can be read in and immediately cast to one of the four types above. The biggest advantage of this change is mixed-type arithmetic is now fully supported.
There is a similar change to the complex type. There are now two "hard-wired" complex types: `complex`, a single precision complex type, and `dcomplex`, a double precision complex type. In order to support the various other complex types, there is a limited template-instantiated complex type, `Complex<`*type*`>`, which can be used to read in data and then cast to one of the two hard-wired types.
As with the signal classes, mixed-type arithmetic is supported between all complex types.

B References

References

[1] Margaret A. Ellis and Bjarne Stroustrup, The Annotated C++ Reference Manual, Addison-Wesley publishing Company, 1990, ISBN 0-201-51459-1

[2] Bjarne Stroustrup, The C++ Programming Language, 2ed, Addison-Wesley publishing Company, 1994, ISBN 0-201-53992-6

[3] Hanna Szoke, James M. Brase, Victoria J. Miller, Melvin G. Wieting, *The VIEW Signal and Image Processing System*, Lawrence Livermore National Laboratory, UCID-21368, March 1988

[4] I. Pitas and A. N. Venetsanopoulos, Nonlinear Digital Filters – Principles and Applications, Kluwer Academic Publishers, 1990, ISBN 0-7923-9049-0

[5] I. Pitas and A. N. Venetsanopoulos, *Order Statistics in Digital Image Processing*, Proceedings of the IEEE, December 1992

[6] K. Fukunaga, Introduction to Statistical Pattern Recognition, 2ed, Chap 6, Academic Press, INC., 1990, ISBN 0-12-269851-7

[7] M. C. Axelrod and J. G. Glosup, *Class A Modeling of Narrowband Ocean Acoustic Noise: Theory, Parameter Estimation and Data Analysis*, UCRL-JC-118216, Lawrence Livermore National Laboratory, 1994, submitted to U.S. Navy Journal of underwater acoustics

[8] Norman Brenner, MIT Lincoln Laboratory, IEEE Audio Transactions, Special Issue on FFT, June 1967

[9] E. M. Johansson and J. P. Fitch, *Remarks on Translation by Sign Modulation*, IEEE Trans. Signal Processing, April 1991

Index

[], 30, 147

AppendToSPRFile, 38, 179
arg, 50, 161
array
  [], 147
  accessing elements, 147
  casts, 148
  number of elements, 151
ascii, 40
assignment operators
  array, 146
  complex, 155
  from a file to a signal, 26
  from a scalar to a signal, 27
  signal, 25
  string, 175
atmean, 101
autocorrelate, 55
average, 55

barrel shift, 56
BARTLETT, 144, 199
bartlett, 129
BLACKMAN, 144, 199
blackman, 130
bshift, 56

casts
  array, 148
  complex, 159
  signal, 30
  strings, 174
cbrt, 161
cc, 50, 162
center of mass, 59
centerpad, 105
central moments, 57
CHAR, 10, 197
classArandom, 113
clip, 56
cmoment, 57
col_fft, 91
col_hfft, 93
com, 59
COMPLEX, 10, 197
complex, 153
  casts, 159
  constructors, 153
  type definitions, 171
complex conjugate
  complex signal, 50
  complex variable, 162
complex exponential
  complex variable, 163
  signal, 51
constant, 37
constructor
  array, 145
    copy, 145
    default, 145
    from a pointer, 146
  complex, 153
    copy, 154
    data, 154
    default, 153
  signal, 15
    copy, 17
    default, 15
    detailed, 18
    file, 23
    mixed, 24
    size, 21
  string, 173
    copy, 173
    default, 173
    from a char*, 173
    from a char, 174
    from an int, 174
convolve, 59
correlate, 59
cos, 50, 162
cosine
  complex variable, 162
  signal, 50
crosscorrelate, 60
cube root
  complex variable, 161

DCOMPLEX, 10, 197
DeleteViewSignal, 180
demean, 60
derivative
  centered approximation to first, 61
  higher order approximations to first, 73
det, 124
determinant, 124
detrend, 60
differentiate, 61
dimof, 32, 180
dirof, 35
DOUBLE, 10, 197

enorm, 123
etmean, 102
Euclidean norm, 123
euler, 51, 163
Euler exponential
    complex variable, 163
    signal, 51
evaluate, 108, 196
exp, 51, 165
exponential
    complex variable, 165
    signal, 51
exprrandom, 114
extract
    from a signal on disk, 23
extract
    from a signal in memory, 62

fabs
    complex variable, 165
    signal, 51
FAILURE, 199
fft, 94
FFT_HP, 89, 198
FFT_NM, 89, 198
FFT_SN, 89, 198
FFT_SP, 89, 198
FFT_TA, 89, 198
FFT_TB, 89, 198
flatten, 70
FLOAT, 10, 197
fold, 70
Fourier transforms, 89
    column, 91
    FFT_HP, 89
    FFT_NM, 89
    FFT_SN, 89
    FFT_SP, 89
    FFT_TA, 89
    FFT_TB, 89
    full, 94
    half-plane, 96
    half-plane column, 93
    half-plane row, 99
    row, 97
function generators, 129
    Bartlett window, 129
    Blackman window, 130
    box, 133
    chirp, 134
    cosine, 136
    disk, 135
    ellipse, 137
    Euler exponential, 138
    filter window, 144
    Gaussian, 140
    general, 139
    Hamming window, 131
    Hanning window, 132
    ramp, 141
    sine, 142
    sphere, 135
    square wave, 143

gammarandom, 115, 195
GAUSSIAN, 144, 199
gaussianrandom, 117, 196
gaxmy, 124
gaxpy, 124
getbase, 178
getcol, 47
getdir, 178
getpt, 46, 193
getrow, 47, 48
getting rows and columns, 47
    copying a column from a signal to memory, 47
    copying a row from a signal to memory, 48
    getting a column from a signal returning a signal, 47
    getting a row from a signal returning a signal, 47
gettype, 36
GoldenRatio, 199

HAMMING, 144, 199
hamming, 131
HANNING, 144, 199
hanning, 132
hfft, 96
histogram, 63

identity, 121
imag, 52, 158
imaginary part
    complex signal, 52
    complex variable, 158
inner, 123
insert, 63
INT, 10, 197
integrate, 64
intensity
    complex variable, 166
    signal, 52
intensity, 52, 166
interpolate, 110
intervalof, 33, 181
inverse, 122

ipdemean, 60
ipflatten, 70
ipfold, 70
ipinverse, 122
ipnormalize, 71
ipshsort, 44
ipsolve, 122
iptranspose, 126
isidp, 181

Krandom, 118

local mean, 65
localmean, 65
localstat, 67
log, 53, 166
log10, 53, 166
LONG, 10, 197

magnify, 68
magnitude
    complex variable, 165
    signal, 51
magnitude and phase conversion
    complex variable, 167
    signal, 52
math functions
    complex, 161
    signal, 50
matrix, 121
    determinant, 124
    equation solve, 122
    Euclidean norm, 123
    formated printing, 125
    gaxmy, 124
    gaxpy, 124
    identity, 121
    inverse, 122
    matrix-matrix multiply, 126
    transpose, 126
    vector inner product, 123
    weighted vector inner product, 127
matrixform, 125
MAX, 102, 194
MAX_DATA_TYPE, 197
MAXDIM, 197
maximum, 36
mean removal, 60
median, 103, 194
merge, 68
MIN, 103, 194
MIN_DATA_TYPE, 197
minimum, 36
mkbox, 133
mkchirp, 134
mkcircle, 135
mkcosine, 136
mkellipse, 137
mkeuler, 138
mkfield, 139
mkgauss, 140
mkramp, 141
mksine, 142
mksqrwv, 143
mkwindow, 144
mmm, 126
moments
    central, 57
    first two, 69
moments, 69
mptori
    complex variable, 167
    signal, 52

nameof, 34
ndifferentiate, 73
ndimof, 31, 180
nelemof, 31, 151
NORM_TO_MAX, 71, 198
NORM_TO_MAXMAG, 71, 198
NORM_TO_MEAN, 71, 198
NORM_TO_SUM, 71, 198
NORM_TO_SUMFABS, 71, 198
NORM_TO_VAR, 71, 198
normalize, 71

OpenView, 182
order statistic filters, 101
    α-trimmed mean, 101
    edge-trimmed mean, 102
    max filter, 102
    median filter, 103
    min filter, 103
originof, 32, 183
OTHER, 199
overlay, 73

padding, 105
    center, 105
    wrapping, 105
    zero, 106
parse a UNIX path, 178
PDF, 74
pdf, 75
pdfh, 76
pdfi, 75
phase
    complex signal, 50

complex variable, 161
Pi, 199
polar, 167
pow, 53, 168
powerspectrum, 79
probability density function, 74
project, 79
ptrto, 34
putcol, 49
putrow, 49

R_BILINEAR, 68, 198
R_REPLICATE, 68, 198
R_SPLINE, 68, 198
R_ZEROINS, 68, 198
ran1, 195
random number generator, 113
Class A, 113
exponential, 114
Gamma, 115
Gaussian, 117
K-distributed, 118
uniform, 120
ReadSPRFile, 183
ReadView, 185
ReadViewColumn, 186
ReadViewRow, 187
real, 53, 158
real part
complex signal, 53
complex variable, 158
rect, 168
rect_x, 169
rect_y, 169
RECTANGULAR, 144, 199
remove, 39
replicate, 79
resample, 80
ResolveName, 188
rotate, 80
row_fft, 97
row_hfft, 99

save, 38
SaveAsView, 189
shift
barrel, 56
zero-insert, 81
shift, 81
SHORT, 10, 197
show, 171
show
complex, 171
signal, 42
string, 177
shsort, 44, 195
signal
[], 30
accessing elements, 30
appending comments to SPR file, 38
array of, 16, 20, 22, 23
assignment operators, 25
from a file, 26
from a scalar, 27
axis intervals, 33
axis origins, 32
axis sizes, 32
casts, 30
constructor, 15
copy, 17
default, 15
detailed, 18
file, 23
mixed, 24
size, 21
data type, 35
deleting from disk, 39
directory, 35
fill with a constant, 37
getting and setting elements, 46
getting rows and columns, 47
listing in ASCII, 40
maximum, 36
minimum, 36
name, 34
name of data type, 36
number of dimensions, 31
number of elements, 31
pointer to data member, 30, 34
properties, 14
saving, 38
showing information about, 42
sorting elements, 44
storing rows and columns, 49
sum, 37
sin, 54, 169
sine
complex variable, 169
signal, 54
smooth, 81
snr, 83
solve, 122
sorting, 44
spectrogram, 85
spline, 107
computation of coefficients, 111
evaluation of, 108

interpolate, 110
spline, 111
sqrt, 54, 170
statistics
global, 86
local, 67
statistics, 86
storing rows and columns, 49
inserting a 1D signal into a column of 2D signal, 49
inserting a 1D signal into a row of 2D signal, 49
strings, 173
casts, 174
parse a UNIX path, 178
showing contents, 177
stypeof, 35, 190
SUCCESS, 199
sum
signal elements, 37
signal intensities, 82
signal magnitudes, 82
sum, 37
sumfabs, 82
sumint, 82

T_AXIS, 197
threshold, 88
transpose, 126
TypeReadView, 184

UCCOMPLEX, 10, 197
UCHAR, 10, 197
uniformrandom, 120, 195

vector
equation solve, 122
Euclidean norm, 123
formated printing, 125
gaxmy, 124
gaxpy, 124
inner product, 123
weighted inner product, 127
View support, 179
appending comments to SPR file, 179
deleting signal from disk, 180
determining if a file is a View file, 181
opening View files, 182
reading axis intervals, 181
reading axis origins, 183
reading axis sizes, 180
reading data file, 184
reading parameter file, 183
reading rows and columns, 186
reading SDT file, 184
reading signal dimensionality, 180
reading signal type, 190
reading SPR file, 183
resolving a file name, 188
saving, 189
writing parameters, 191
writing SPR file, 191

WRA_COOR, 40, 199
WRA_INDEX, 40, 199
WRA_NONE, 40, 199
wrappad, 105
WriteSPRFile, 191

X_AXIS, 197
xtay, 127

Y_AXIS, 197

Z_AXIS, 197
zeropad, 106