FreeMat v2.0 Documentation

Samit Basu

June 5, 2006

Contents

Chapter 1

Introduction and Getting Started

1.1 INSTALL Installing FreeMat

1.1.1 General Instructions

Here are the general instructions for installing FreeMat. First, follow the instructions listed below for the platform of interest. Then, run the

```
-->pathtool
```

which brings up the path setup tool. More documentation on the GUI elements (and how to use them) will be forthcoming.

1.1.2 Linux

For Linux, FreeMat is now provided as a binary installation. To install it simply download the binary using your web browser, and then unpack it

```
tar xvfz FreeMat-2.0-Linux-Binary.tar.gz
```

You can then run FreeMat directly without any additional effort

```
FreeMat-2.0-Linux-Binary/Contents/bin/FreeMat
```

will start up FreeMat as an X application. If you want to run it as a command line application (to run from within an xterm), use the `nogui` flag

```
FreeMat-2.0-Linux-Binary/Contents/bin/FreeMat -nogui
```

If you do not want FreeMat to use X at all (no graphics at all), use the `noX` flag

```
FreeMat-2.0-Linux-Binary/Contents/bin/FreeMat -noX
```

For convenience, you may want to add FreeMat to your path. The exact mechanism for doing this depends on your shell. Assume that you have unpacked `FreeMat-2.0-Linux-Binary.tar.gz` into the directory `/home/myname`. Then if you use `csh` or its derivatives (like `tcsh`) you should add the following line to your `.cshrc` file:

```
set path=($path /home/myname/FreeMat-2.0-Linux/Binary/Contents/bin)
```

If you use `bash`, then add the following line to your `.bash_profile`

```
PATH=$PATH:/home/myname/FreeMat-2.0-Linux/Binary/Contents/bin
```

If the prebuilt binary package does not work for your Linux distribution, you will need to build FreeMat from source (see the source section below). When you have FreeMat running, you can setup your path using the `pathtool`. Note that the `FREEMAT_PATH` is no longer used by FreeMat. You must use the `pathtool` to adjust the path.

1.1.3 Windows

For Windows, FreeMat is installed via a binary installer program. To use it, simply download the setup program `FreeMat-2.0-Setup.exe`, and double click it. Follow the instructions to do the installation, then setup your path using `pathtool`.

1.1.4 Mac OS X

For Mac OS X, FreeMat is distributed as an application bundle. To install it, simply download the compressed disk image file `FreeMat-2.0.dmg`, double click to mount the disk image, and then copy the application `FreeMat-2.0` to some convenient place. To run FreeMat, simply double click on the application. Run `pathtool` to setup your FreeMat path.

1.1.5 Source Code

The source code build is a little more complicated than previous versions of FreeMat. Here are the current build instructions for all platforms.

1. Build and install Qt 4.1 or later - `http://www.trolltech.com/download/opensource.html`
2. Install g77 or gfortran (use fink for Mac OS X, use `gcc-g77` package for MinGW)
3. Download the source code `FreeMat-2.0-src.tar.gz`.
4. Unpack the source code: `tar xvfz FreeMat-2.0-src.tar.gz`.
5. For Windows, you will need to install MSYS as well as MINGW tobuild FreeMat. You will also need unzip to unpack the enclosedmatio.zip archive
6. If you are extraordinarily lucky (or prepared), you can issue theusual `configure && make && make ins` This is not likely to workbecause of the somewhat esoteric dependencies of FreeMat. The configurestep will probably fail and indicate what external dependencies arestill needed. It will also create a script that you can run to buildthe missing dependencies.

For example, on my machine, a `configure` step yields the following result:

```
checking for amd_postorder in -lamd... no
checking for umfpack_zl_solve in -lumfpack... no
checking for fftwf_malloc in -lfftw3f... no
checking for fftw_malloc in -lfftw3... no
checking for sgemm_... no
checking for ATL_xerbla in -latlas... no
checking for sgemm_ in -lblas... no
checking for sgemm_ in -lcxml... no
checking for sgemm_ in -ldxml... no
checking for sgemm_ in -lscs... no
checking for sgemm_ in -lcomplib.sgimath... no
checking for sgemm_ in -lblas... (cached) no
checking for sgemm_ in -lm... no
checking for sgemm_ in -lblas... (cached) no
checking for znaupd_ in -larpack... no
checking for inflate in -lz... yes
checking for Mat_Open in -lmatio... no
configure: creating ./config.status
config.status: creating tools/disttool/builddeps
config.status: executing depfiles commands
configure: error:
*********************************************************************
One or more of the following external dependencies was not
found:

  AMD                        no
  UMFPACK                    no
  FFTW3 (Single Precision)  no
  FFTW3 (Double Precision)  no
  BLAS                       no
  LAPACK                     no
  ARPACK                     no
  ZLIB                       yes
  MATIO                      no
*********************************************************************
A script to build these external dependencies has been created
in the current directory.  To build the missing dependencies,
run the script via:

./builddeps  --with-ffcall --with-umfpack --with-umfpack --with-fftw --with-fftw   --with-blas --with-la

Note that this will attempt to build and install the libraries
and header files in extern/Root/lib and extern/Root/include
(respectively).  Once the required libraries have been successfully
built, rerun configure.
```

After you run the `builddeps` script, the configure succeeds, and the usual `configure && make && make inst` should work. Note that for Linux, the location of Qt4 is highly system dependent. For `configure` to find the whereabouts of Qt4, you need to make sure that `pkg-config` can find Qt4. For example, if you installed Qt4 yourself, you would set

```
declare -x PKG_CONFIG_PATH=/usr/local/Trolltech/Qt-4.1.0/lib
```

Also, to build a binary distributable (app bundle on the Mac, setup installer on win32, and a binary distribution on Linux), you will need to run `make package` instead of `make install`.

Chapter 2

Variables and Arrays

2.1 CELL Cell Array Definitions

2.1.1 Usage

The cell array is a fairly powerful array type that is available in FreeMat. Generally speaking, a cell array is a heterogenous array type, meaning that different elements in the array can contain variables of different type (including other cell arrays). For those of you familiar with `C`, it is the equivalent to the `void *` array. The general syntax for their construction is

```
   A = {row_def1;row_def2;...;row_defN}
```

where each row consists of one or more elements, seperated by commas

```
  row_defi = element_i1,element_i2,...,element_iM
```

Each element can be any type of FreeMat variable, including matrices, arrays, cell-arrays, structures, strings, etc. The restriction on the definition is that each row must have the same number of elements in it.

2.1.2 Examples

Here is an example of a cell-array that contains a number, a string, and an array

```
--> A = {14,'hello',[1:10]}
A =
  <cell array> - size: [1 3]

Columns 1 to 3
 [14]    hello    [[1 10] int32]
```

Note that in the output, the number and string are explicitly printed, but the array is summarized. We can create a 2-dimensional cell-array by adding another row definition

```
--> B = {pi,i;e,-1}
B =
  <cell array> - size: [2 2]

Columns 1 to 2
 [3.141593]    [0.000000+1.000000i]
 [2.718282]    [-1]
```

Finally, we create a new cell array by placing `A` and `B` together

```
--> C = {A,B}
C =
  <cell array> - size: [1 2]

Columns 1 to 2
 {[1 3] cell }    {[2 2] cell }
```

2.2 Function Handles

2.2.1 Usage

Starting with version 1.11, FreeMat now supports `function handles`, or `function pointers`. A `function handle` is an alias for a function or script that is stored in a variable. First, the way to assign a function handle is to use the notation

```
handle = @func
```

where `func` is the name to point to. The function `func` must exist at the time we make the call. It can be a local function (i.e., a subfunction). To use the `handle`, we can either pass it to `feval` via

```
[x,y] = feval(handle,arg1,arg2).
```

Alternately, you can the function directly using the notation

```
[x,y] = handle(arg1,arg2)
```

2.3 GLOBAL Global Variables

2.3.1 Usage

Global variables are shared variables that can be seen and modified from any function or script that declares them. The syntax for the `global` statement is

```
global variable_1 variable_2 ...
```

The `global` statement must occur before the variables appear.

2.3.2 Example

Here is an example of two functions that use a global variable to communicate an array between them. The first function sets the global variable.

```
     set_global.m
function set_global(x)
  global common_array
  common_array = x;
```

The second function retrieves the value from the global variable

```
     get_global.m
function x = get_global
  global common_array
  x = common_array;
```

Here we exercise the two functions

```
--> set_global('Hello')
--> get_global
ans =
  <string>  - size: [1 5]
 Hello
```

2.4 INDEXING Indexing Expressions

2.4.1 Usage

There are three classes of indexing expressions available in FreeMat: `()`, `{}`, and `.` Each is explained below in some detail, and with its own example section.

2.4.2 Array Indexing

We start with array indexing `()`, which is the most general indexing expression, and can be used on any array. There are two general forms for the indexing expression - the N-dimensional form, for which the general syntax is

```
  variable(index_1,index_2,...,index_n)
```

and the vector form, for which the general syntax is

```
  variable(index)
```

Here each index expression is either a scalar, a range of integer values, or the special token `:`, which is shorthand for `1:end`. The keyword `end`, when included in an indexing expression, is assigned the length of the array in that dimension. The concept is easier to demonstrate than explain. Consider the following examples:

```
--> A = zeros(4)
A =
  <float>  - size: [4 4]

Columns 1 to 4
 0  0  0  0
 0  0  0  0
 0  0  0  0
 0  0  0  0
--> B = float(randn(2))
B =
  <float>  - size: [2 2]

Columns 1 to 2
  0.74762058   0.43875891
 -0.84486866  -0.56751561
--> A(2:3,2:3) = B
A =
  <float>  - size: [4 4]

Columns 1 to 4
  0.00000000   0.00000000   0.00000000   0.00000000
  0.00000000   0.74762058   0.43875891   0.00000000
  0.00000000  -0.84486866  -0.56751561   0.00000000
  0.00000000   0.00000000   0.00000000   0.00000000
```

Here the array indexing was used on the left hand side only. It can also be used for right hand side indexing, as in

```
--> C = A(2:3,1:end)
C =
  <float>  - size: [2 4]

Columns 1 to 4
  0.00000000   0.74762058   0.43875891   0.00000000
  0.00000000  -0.84486866  -0.56751561   0.00000000
```

Note that we used the `end` keyword to avoid having to know that `A` has 4 columns. Of course, we could also use the `:` token instead:

```
--> C = A(2:3,:)
C =
  <float>  - size: [2 4]

Columns 1 to 4
  0.00000000   0.74762058   0.43875891   0.00000000
  0.00000000  -0.84486866  -0.56751561   0.00000000
```

An extremely useful example of `:` with array indexing is for slicing. Suppose we have a 3-D array, that is `2 x 2 x 3`, and we want to set the middle slice:

```
--> D = zeros(2,2,3)
D =
  <float>  - size: [2 2 3]
(:,:,1) =

Columns 1 to 2
 0  0
 0  0
(:,:,2) =

Columns 1 to 2
 0  0
 0  0
(:,:,3) =

Columns 1 to 2
 0  0
 0  0
--> D(:,:,2) = int32(10*rand(2,2))
D =
  <float>  - size: [2 2 3]
(:,:,1) =

Columns 1 to 2
 0  0
 0  0
(:,:,2) =

Columns 1 to 2
 6  1
 2  7
(:,:,3) =

Columns 1 to 2
 0  0
 0  0
```

In another level of nuance, the assignment expression will automatically fill in the indexed rectangle on the left using data from the right hand side, as long as the lengths match. So we can take a vector and roll it into a matrix using this approach:

```
--> A = zeros(4)
A =
  <float>  - size: [4 4]
```

```
Columns 1 to 4
 0  0  0  0
 0  0  0  0
 0  0  0  0
 0  0  0  0
--> v = [1;2;3;4]
v =
  <int32>  - size: [4 1]

Columns 1 to 1
 1
 2
 3
 4
--> A(2:3,2:3) = v
A =
  <float>  - size: [4 4]

Columns 1 to 4
 0  0  0  0
 0  1  3  0
 0  2  4  0
 0  0  0  0
```

The N-dimensional form of the variable index is limited to accessing only (hyper-) rectangular regions of the array. You cannot, for example, use it to access only the diagonal elements of the array. To do that, you use the second form of the array access (or a loop). The vector form treats an arbitrary N-dimensional array as though it were a column vector. You can then access arbitrary subsets of the arrays elements (for example, through a `find` expression) efficiently. Note that in vector form, the `end` keyword takes the meaning of the total length of the array (defined as the product of its dimensions), as opposed to the size along the first dimension.

2.4.3 Cell Indexing

The second form of indexing operates, to a large extent, in the same manner as the array indexing, but it is by no means interchangable. As the name implies, `cell`-indexing applies only to `cell` arrays. For those familiar with `C`, cell- indexing is equivalent to pointer derefencing in `C`. First, the syntax:

```
variable{index_1,index_2,...,index_n}
```

and the vector form, for which the general syntax is

```
variable{index}
```

The rules and interpretation for N-dimensional and vector indexing are identical to `()`, so we will describe only the differences. In simple terms, applying `()` to a cell-array returns another cell array that is a subset of the original array. On the other hand, applying {} to a cell-array returns the contents of that cell array. A simple example makes the difference quite clear:

```
--> A = {1, 'hello', [1:4]}
A =
  <cell array> - size: [1 3]

Columns 1 to 3
 [1]    hello    [[1 4] int32]
--> A(1:2)
ans =
  <cell array> - size: [1 2]

Columns 1 to 2
 [1]    hello
--> A{1:2}
ans =

1 of 2:
  <int32>  - size: [1 1]
 1

2 of 2:
  <string>  - size: [1 5]
 hello
```

You may be surprised by the response to the last line. The output is multiple assignments to `ans`!. The output of a cell-array dereference can be used anywhere a list of expressions is required. This includes arguments and returns for function calls, matrix construction, etc. Here is an example of using cell-arrays to pass parameters to a function:

```
--> A = {[1,3,0],[5,2,7]}
A =
  <cell array> - size: [1 2]

Columns 1 to 2
 [[1 3] int32]    [[1 3] int32]
--> max(A{1:end})
ans =
  <int32>  - size: [1 3]

Columns 1 to 3
 5  3  7
```

And here, cell-arrays are used to capture the return.

```
--> [K{1:2}] = max(randn(1,4))
K =
```

```
  <cell array> - size: [1 2]

Columns 1 to 2
 [1.339117]    [4]
```

Here, cell-arrays are used in the matrix construction process:

```
--> C = [A{1};A{2}]
C =
  <int32>  - size: [2 3]

Columns 1 to 3
 1  3  0
 5  2  7
```

Note that this form of indexing is used to implement variable length arguments to function. See `varargin` and `varargout` for more details.

2.4.4 Structure Indexing

The third form of indexing is structure indexing. It can only be applied to structure arrays, and has the general syntax

```
  variable.fieldname
```

where `fieldname` is one of the fields on the structure. Note that in FreeMat, fields are allocated dynamically, so if you reference a field that does not exist in an assignment, it is created automatically for you. If variable is an array, then the result of the `.` reference is an expression list, exactly like the `{}` operator. Hence, we can use structure indexing in a simple fashion:

```
--> clear A
--> A.color = 'blue'
A =
  <structure array> - size: [1 1]
    color: blue
--> B = A.color
B =
  <string>  - size: [1 4]
 blue
```

Or in more complicated ways using expression lists for function arguments

```
--> clear A
--> A(1).maxargs = [1,6,7,3]
A =
  <structure array> - size: [1 1]
    maxargs: [[1 4] int32]
--> A(2).maxargs = [5,2,9,0]
A =
```

```
  <structure array> - size: [1 2]
  Fields
    maxargs
--> max(A.maxargs)
ans =
  <int32>  - size: [1 4]

Columns 1 to 4
 5  6  9  3
```

or to store function outputs

```
--> clear A
--> A(1).maxreturn = [];
--> A(2).maxreturn = [];
--> [A.maxreturn] = max(randn(1,4))
A =
  <structure array> - size: [1 2]
  Fields
    maxreturn
```

FreeMat now also supports the so called dynamic-field indexing expressions. In this mode, the fieldname is supplied through an expression instead of being explicitly provided. For example, suppose we have a set of structure indexed by color,

```
--> x.red = 430;
--> x.green = 240;
--> x.blue = 53;
--> x.yello = 105
x =
  <structure array> - size: [1 1]
    red: [430]
    green: [240]
    blue: [53]
    yello: [105]
```

Then we can index into the structure `x` using a dynamic field reference:

```
--> y = 'green'
y =
  <string>  - size: [1 5]
 green
--> a = x.(y)
a =
  <int32>  - size: [1 1]
 240
```

Note that the indexing expression has to resolve to a string for dynamic field indexing to work.

2.4.5 Complex Indexing

The indexing expressions described above can be freely combined to affect complicated indexing expressions. Here is an example that exercises all three indexing expressions in one assignment.

```
--> Z{3}.foo(2) = pi
Z =
  <cell array> - size: [1 3]

Columns 1 to 3
 []    []     [1 1] struct array
```

From this statement, FreeMat infers that Z is a cell-array of length 3, that the third element is a structure array (with one element), and that this structure array contains a field named 'foo' with two double elements, the second of which is assigned a value of pi.

2.5 MATRIX Matrix Definitions

2.5.1 Usage

The matrix is the basic datatype of FreeMat. Matrices can be defined using the following syntax

```
  A = [row_def1;row_def2;...,row_defN]
```

where each row consists of one or more elements, seperated by commas

```
  row_defi = element_i1,element_i2,...,element_iM
```

Each element can either be a scalar value or another matrix, provided that the resulting matrix definition makes sense. In general this means that all of the elements belonging to a row have the same number of rows themselves, and that all of the row definitions have the same number of columns. Matrices are actually special cases of N-dimensional arrays where `N<=2`. Higher dimensional arrays cannot be constructed using the bracket notation described above. The type of a matrix defined in this way (using the bracket notation) is determined by examining the types of the elements. The resulting type is chosen so no information is lost on any of the elements (or equivalently, by choosing the highest order type from those present in the elements).

2.5.2 Examples

Here is an example of a matrix of `int32` elements (note that untyped integer constants default to type `int32`).

```
--> A = [1,2;5,8]
A =
  <int32>  - size: [2 2]

Columns 1 to 2
 1  2
 5  8
```

Now we define a new matrix by adding a column to the right of `A`, and using float constants.

```
--> B = [A,[3.2f;5.1f]]
B =
  <float>  - size: [2 3]

Columns 1 to 3
 1.0  2.0  3.2
 5.0  8.0  5.1
```

Next, we add extend `B` by adding a row at the bottom. Note how the use of an untyped floating point constant forces the result to be of type `double`

```
--> C = [B;5.2,1.0,0.0]
C =
  <double>  - size: [3 3]

Columns 1 to 3
 1.000000000000000  2.000000000000000  3.200000047683716
 5.000000000000000  8.000000000000000  5.099999904632568
 5.200000000000000  1.000000000000000  0.000000000000000
```

If we instead add a row of `complex` values (recall that `i` is a `complex` constant, not a `dcomplex` constant)

```
--> D = [B;2.0f+3.0f*i,i,0.0f]
D =
  <complex>  - size: [3 3]

Columns 1 to 3
 1.00.0 i  2.00.0 i  3.20.0 i
 5.00.0 i  8.00.0 i  5.10.0 i
 2.0+3.0 i  0.0+1.0 i  0.00.0 i
```

Likewise, but using `dcomplex` constants

```
--> E = [B;2.0+3.0*i,i,0.0]
E =
  <dcomplex>  - size: [3 3]

Columns 1 to 2
 1.0000000000000000.000000000000000i  2.0000000000000000.000000000000000i
 5.0000000000000000.000000000000000i  8.0000000000000000.000000000000000i
 2.000000000000000+3.000000000000000i  0.000000000000000+1.000000000000000i

Columns 3 to 3
 3.2000000476837160.000000000000000i
 5.0999999046325680.000000000000000i
 0.0000000000000000.000000000000000i
```

Finally, in FreeMat, you can construct matrices with strings as contents, but you have to make sure that if the matrix has more than one row, that all the strings have the same length.

```
--> F = ['hello';'there']
F =
  <string>  - size: [2 5]
 hello
 there
```

2.6 PERSISTENT Persistent Variables

2.6.1 Usage

Persistent variables are variables whose value persists between calls to a function or script. The general syntax for its use is

```
   persistent variable1 variable2 ... variableN
```

The `persistent` statement must occur before the variable is the tagged as persistent.

2.6.2 Example

Here is an example of a function that counts how many times it has been called.

```
     count_calls.m
function count_calls
  persistent ccount
  if (isempty(ccount)) ccount = 0; end;
  ccount = ccount + 1;
  printf('Function has been called %d times\n',ccount);
```

We now call the function several times:

```
--> for i=1:10; count_calls; end
Function has been called 1 times
Function has been called 2 times
Function has been called 3 times
Function has been called 4 times
Function has been called 5 times
Function has been called 6 times
Function has been called 7 times
Function has been called 8 times
Function has been called 9 times
Function has been called 10 times
```

2.7 STRING String Arrays

2.7.1 Usage

FreeMat supports a `string` array type that operates very much as you would expect. Strings are stored internally as 8-bit values, and are otherwise similar to numerical arrays in all respects. In some respects, this makes strings arrays less useful than one might imagine. For example, numerical arrays in 2-D are rectangular. Thus, each row in the array must have the same number of columns. This requirement is natural for numerical arrays and matrices, but consider a string array. If one wants to store multiple strings in a single data structure, they must all be the same length (unlikely). The alternative is to use a cell array of strings, in which case, each string can be of arbitrary length. Most of the functions that support strings in a set-theoretic way, like `unique` and `sort` operate on cell-arrays of strings instead of string arrays. Just to make the example concrete, here is the old way of storing several strings in an array:

```
--> % This is an error
--> A = ['hello';'bye']
Error: Mismatch in dimension for rows in matrix definition
--> % This is OK, but awkward
--> A = ['hello';'bye  ']
A =
  <string>  - size: [2 5]
 hello
 bye
--> % This is the right way to do it
--> A = {'hello','bye'}
A =
  <cell array> - size: [1 2]

Columns 1 to 2
 hello    bye
```

One important (tricky) point in FreeMat is the treatment of escape sequences. Recall that in `C` programming, an escape sequence is a special character that causes the output to do something unusual. FreeMat supports the following escape sequences:

- `\t` - causes a tab to be output
- `\r` - causes a carriage return (return to the beginning of the line of output, and overwrite the text)
- `\n` - causes a linefeed (advance to next line)

FreeMat follows the `Unix/Linux` convention, that a `\n` causes both a carriage return and a linefeed. To put a single quote into a string use the MATLAB convention of two single quotes, not the `\'` sequence. Here is an example of a string containing some escape sequences:

```
--> a = 'I can''t use contractions\n\tOr can I?\n'
a =
```

```
  <string>  - size: [1 39]
 I can't use contractions\n\tOr can I?\n
```

Now, note that the string itself still contains the `\n` characters. With the exception of the `\'`, the escape sequences do not affect the output unless the strings are put through `printf` or `fprintf`. For example, if we `printf` the variable `a`, we see the `\n` and `\t` take effect:

```
--> printf(a);
I can't use contractions
Or can I?
```

The final complicating factor is on `MSWin` systems. There, filenames regularly contain \ characters. Thus, if you try to print a string containing the filename `C:\redball\timewarp\newton.txt`, the output will be mangled because FreeMat thinks the `\r`, `\t` and `\n` are escape sequences. You have two options. You can use `disp` to show the filename (`disp` does not do escape translation to be compatible with MATLAB). The second option is to escape the backslashes in the string, so that the string you send to `printf` contains `C:\\redball\\timewarp\\newton.txt`.

```
--> % disp displays it ok
--> a = 'C:\redball\timewarp\newton.txt'
a =
  <string>  - size: [1 30]
 C:\redball\timewarp\newton.txt
--> % printf makes a mess
--> printf(a)
C:
edball imewarp
ewton.txt--> % If we double up the slashes it works fine
--> a = 'C:\\redball\\timewarp\\newton.txt'
a =
  <string>  - size: [1 33]
 C:\\redball\\timewarp\\newton.txt
--> printf(a)
C:\redball\timewarp\newton.txt
```

2.8 STRUCT Structure Array Constructor

2.8.1 Usage

Creates an array of structures from a set of field, value pairs. The syntax is

```
   y = struct(n1,v1,n2,v2,...)
```

where `ni` are the names of the fields in the structure array, and `vi` are the values. The values `v_i` must either all be scalars, or be cell-arrays of all the same dimensions. In the latter case, the output structure array will have dimensions dictated by this common size. Scalar entries for the `v_i` are replicated to fill out their dimensions. An error is raised if the inputs are not properly matched (i.e., are not pairs of field names and values), or if the size of any two non-scalar values cell-arrays are different.

Another use of the `struct` function is to convert a class into a structure. This allows you to access the members of the class, directly but removes the class information from the object.

2.8.2 Example

This example creates a 3-element structure array with two fields, `foo` and `bar`, where the contents of `foo` are provided explicitly, and the contents of `bar` are replicated from a scalar.

```
--> y = struct('foo',{1,3,4},'bar',{'cheese','cola','beer'},'key',508)
y =
  <structure array> - size: [1 3]
  Fields
    foo
    bar
    key
--> y(1)
ans =
  <structure array> - size: [1 1]
    foo: [1]
    bar: cheese
    key: [508]
--> y(2)
ans =
  <structure array> - size: [1 1]
    foo: [3]
    bar: cola
    key: [508]
--> y(3)
ans =
  <structure array> - size: [1 1]
    foo: [4]
    bar: beer
    key: [508]
```

Chapter 3

Functions and Scripts

3.1 FUNCTION Function Declarations

3.1.1 Usage

There are several forms for function declarations in FreeMat. The most general syntax for a function declaration is the following:

```
function [out_1,...,out_M,varargout] = fname(in_1,...,in_N,varargin)
```

where `out_i` are the output parameters, `in_i` are the input parameters, and `varargout` and `varargin` are special keywords used for functions that have variable inputs or outputs. For functions with a fixed number of input or output parameters, the syntax is somewhat simpler:

```
function [out_1,...,out_M] = fname(in_1,...,in_N)
```

Note that functions that have no return arguments can omit the return argument list (of `out_i`) and the equals sign:

```
function fname(in_1,...,in_N)
```

Likewise, a function with no arguments can eliminate the list of parameters in the declaration:

```
function [out_1,...,out_M] = fname
```

Functions that return only a single value can omit the brackets

```
function out_1 = fname(in_1,...,in_N)
```

In the body of the function `in_i` are initialized with the values passed when the function is called. Also, the function must assign values for `out_i` to pass values to the caller. Note that by default, FreeMat passes arguments by value, meaning that if we modify the contents of `in_i` inside the function, it has no effect on any variables used by the caller. Arguments can be passed by reference by prepending an ampersand `&` before the name of the input, e.g.

```
function [out1,...,out_M] = fname(in_1,&in_2,in_3,...,in_N)
```

in which case in_2 is passed by reference and not by value. Also, FreeMat works like C in that the caller does not have to supply the full list of arguments. Also, when keywords (see help keywords) are used, an arbitrary subset of the parameters may be unspecified. To assist in deciphering the exact parameters that were passed, FreeMat also defines two variables inside the function context: nargin and nargout, which provide the number of input and output parameters of the caller, respectively. See help for nargin and nargout for more details. In some circumstances, it is necessary to have functions that take a variable number of arguments, or that return a variable number of results. In these cases, the last argument to the parameter list is the special argument varargin. Inside the function, varargin is a cell-array that contains all arguments passed to the function that have not already been accounted for. Similarly, the function can create a cell array named varargout for variable length output lists. See help varargin and varargout for more details.

The function name fname can be any legal FreeMat identifier. Functions are stored in files with the .m extension. Note that the name of the file (and not the function name fname used in the declaration) is how the function appears in FreeMat. So, for example, if the file is named foo.m, but the declaration uses bar for the name of the function, in FreeMat, it will still appear as function foo. Note that this is only true for the first function that appears in a .m file. Additional functions that appear after the first function are known as helper functions or local functions. These are functions that can only be called by other functions in the same .m file. Furthermore the names of these helper functions are determined by their declaration and not by the name of the .m file. An example of using helper functions is included in the examples.

Another important feature of functions, as opposed to, say scripts, is that they have their own scope. That means that variables defined or modified inside a function do not affect the scope of the caller. That means that a function can freely define and use variables without unintentionally using a variable name reserved elsewhere. The flip side of this fact is that functions are harder to debug than scripts without using the keyboard function, because the intermediate calculations used in the function are not available once the function exits.

3.1.2 Examples

Here is an example of a trivial function that adds its first argument to twice its second argument:

```
     addtest.m
function c = addtest(a,b)
  c = a + 2*b;

--> addtest(1,3)
ans =
  <int32>  - size: [1 1]
 7
--> addtest(3,0)
ans =
  <int32>  - size: [1 1]
 3
```

Suppose, however, we want to replace the value of the first argument by the computed sum. A first attempt at doing so has no effect:

```
     addtest2.m
function addtest2(a,b)
  a = a + 2*b;

--> arg1 = 1
arg1 =
  <int32>  - size: [1 1]
 1
--> arg2 = 3
arg2 =
  <int32>  - size: [1 1]
 3
--> addtest2(arg1,arg2)
--> arg1
ans =
  <int32>  - size: [1 1]
 1
--> arg2
ans =
  <int32>  - size: [1 1]
 3
```

The values of `arg1` and `arg2` are unchanged, because they are passed by value, so that any changes to `a` and `b` inside the function do not affect `arg1` and `arg2`. We can change that by passing the first argument by reference:

```
     addtest3.m
function addtest3(&a,b)
  a = a + 2*b
```

Note that it is now illegal to pass a literal value for `a` when calling `addtest3`:

```
--> addtest3(3,4)
a =
  <int32>  - size: [1 1]
 11
Error: Must have lvalue in argument passed by reference
--> addtest3(arg1,arg2)
a =
  <int32>  - size: [1 1]
 7
--> arg1
ans =
  <int32>  - size: [1 1]
 7
--> arg2
ans =
```

```
  <int32>  - size: [1 1]
 3
```

The first example fails because we cannot pass a literal like the number `3` by reference. However, the second call succeeds, and note that `arg1` has now changed. Note: please be careful when passing by reference - this feature is not available in MATLAB and you must be clear that you are using it.

As variable argument and return functions are covered elsewhere, as are keywords, we include one final example that demonstrates the use of helper functions, or local functions, where multiple function declarations occur in the same file.

```
     euclidlength.m
function y = foo(x,y)
  square_me(x);
  square_me(y);
  y = sqrt(x+y);

function square_me(&t)
  t = t^2;
```

```
--> euclidlength(3,4)
ans =
  <double>  - size: [1 1]
 5
--> euclidlength(2,0)
ans =
  <double>  - size: [1 1]
 2
```

3.2 KEYWORDS Function Keywords

3.2.1 Usage

A feature of IDL that FreeMat has adopted is a modified form of `keywords`. The purpose of `keywords` is to allow you to call a function with the arguments to the function specified in an arbitrary order. To specify the syntax of `keywords`, suppose there is a function with prototype

```
  function [out_1,...,out_M] = foo(in_1,...,in_N)
```

Then the general syntax for calling function `foo` using keywords is

```
  foo(val_1, val_2, /in_k=3)
```

which is exactly equivalent to

```
  foo(val_1, val_2, [], [], ..., [], 3),
```

where the 3 is passed as the k-th argument, or alternately,

```
  foo(val_1, val_2, /in_k)
```

which is exactly equivalent to

```
foo(val_1, val_2, [], [], ..., [], logical(1)),
```

Note that you can even pass reference arguments using keywords.

3.2.2 Example

The most common use of keywords is in controlling options for functions. For example, the following function takes a number of binary options that control its behavior. For example, consider the following function with two arguments and two options. The function has been written to properly use and handle keywords. The result is much cleaner than the MATLAB approach involving testing all possible values of `nargin`, and forcing explicit empty brackets for don't care parameters.

```
     keyfunc.m
function c = keyfunc(a,b,operation,printit)
  if (~isset('a') | ~isset('b'))
    error('keyfunc requires at least the first two 2 arguments');
  end;
  if (~isset('operation'))
    % user did not define the operation, default to '+'
    operation = '+';
  end
  if (~isset('printit'))
    % user did not specify the printit flag, default is false
    printit = 0;
  end
  % simple operation...
  eval(['c = a ' operation ' b;']);
  if (printit)
    printf('%f %s %f = %f\n',a,operation,b,c);
  end
```

Now some examples of how this function can be called using `keywords`.

```
--> keyfunc(1,3)                % specify a and b, defaults for the others
ans =
  <int32>  - size: [1 1]
 4
--> keyfunc(1,3,/printit)       % specify printit is true
1.000000 + 3.000000 = 4.000000
ans =
  <int32>  - size: [1 1]
 4
--> keyfunc(/operation='-',2,3) % assigns a=2, b=3
ans =
  <int32>  - size: [1 1]
 -1
```

```
--> keyfunc(4,/operation='*',/printit) % error as b is unspecified
Error: keyfunc requires at least the first two 2 arguments
In base(base), line 0, column 0
In Eval(keyfunc(4,/operation...), line 1, column 8
In keyfunc(keyfunc), line 3, column 10
```

3.3 NARGIN Number of Input Arguments

3.3.1 Usage

The special variable `nargin` is defined inside of all functions. It indicates how many arguments were passed to the function when it was called. FreeMat allows for fewer arguments to be passed to a function than were declared, and `nargin`, along with `isset` can be used to determine exactly what subset of the arguments were defined. There is no syntax for the use of `nargin` - it is automatically defined inside the function body.

3.3.2 Example

Here is a function that is declared to take five arguments, and that simply prints the value of `nargin` each time it is called.

```
     nargintest.m
function nargintest(a1,a2,a3,a4,a5)
  printf('nargin = %d\n',nargin);
```

```
--> nargintest(3);
nargin = 1
--> nargintest(3,'h');
nargin = 2
--> nargintest(3,'h',1.34);
nargin = 3
--> nargintest(3,'h',1.34,pi,e);
nargin = 5
```

3.4 NARGOUT Number of Output Arguments

3.4.1 Usage

The special variable `nargout` is defined inside of all functions. It indicates how many return values were requested from the function when it was called. FreeMat allows for fewer return values to be requested from a function than were declared, and `nargout` can be used to determine exactly what subset of the functions outputs are required. There is no syntax for the use of `nargout` - it is automatically defined inside the function body.

3.4.2 Example

Here is a function that is declared to return five values, and that simply prints the value of `nargout` each time it is called.

```
     nargouttest.m
function [a1,a2,a3,a4,a5] = nargouttest
  printf('nargout = %d\n',nargout);
  a1 = 1; a2 = 2; a3 = 3; a4 = 4; a5 = 5;
```

```
--> a1 = nargouttest
nargout = 1
a1 =
  <int32>  - size: [1 1]
 1
--> [a1,a2] = nargouttest
nargout = 2
a1 =
  <int32>  - size: [1 1]
 1
a2 =
  <int32>  - size: [1 1]
 2
--> [a1,a2,a3] = nargouttest
nargout = 3
a1 =
  <int32>  - size: [1 1]
 1
a2 =
  <int32>  - size: [1 1]
 2
a3 =
  <int32>  - size: [1 1]
 3
--> [a1,a2,a3,a4,a5] = nargouttest
nargout = 5
a1 =
  <int32>  - size: [1 1]
 1
a2 =
  <int32>  - size: [1 1]
 2
a3 =
  <int32>  - size: [1 1]
 3
a4 =
  <int32>  - size: [1 1]
```

```
 4
a5 =
  <int32>  - size: [1 1]
 5
```

3.5 SCRIPT Script Files

3.5.1 Usage

A script is a sequence of FreeMat commands contained in a `.m` file. When the script is called (via the name of the file), the effect is the same as if the commands inside the script file were issued one at a time from the keyboard. Unlike `function` files (which have the same extension, but have a `function` declaration), script files share the same environment as their callers. Hence, assignments, etc, made inside a script are visible to the caller (which is not the case for functions.

3.5.2 Example

Here is an example of a script that makes some simple assignments and `printf` statements.

```
     tscript.m
a = 13;
printf('a is %d\n',a);
b = a + 32
```

If we execute the script and then look at the defined variables

```
--> tscript
a is 13
b =
  <int32>  - size: [1 1]
 45
--> who
  Variable Name       Type   Flags             Size
              a      int32                    [1 1]
            ans     double                    [0 0]
              b      int32                    [1 1]
             c1     string                    [1 12]
             c2     string                    [1 12]
             c3     string                    [1 14]
         nargin      int32                    [1 1]
        nargout      int32                    [1 1]
      operation     string                    [1 1]
       printit    logical                     [1 1]
```

we see that `a` and `b` are defined appropriately.

3.6 SPECIAL Special Calling Syntax

3.6.1 Usage

To reduce the effort to call certain functions, FreeMat supports a special calling syntax for functions that take string arguments. In particular, the three following syntaxes are equivalent, with one caveat:

```
functionname('arg1','arg2',...,'argn')
```

or the parenthesis and commas can be removed

```
functionname 'arg1' 'arg2' ... 'argn'
```

The quotes are also optional (providing, of course, that the argument strings have no spaces in them)

```
functionname arg1 arg2 ... argn
```

This special syntax enables you to type `hold on` instead of the more cumbersome `hold('on')`. The caveat is that FreeMat currently only recognizes the special calling syntax as the first statement on a line of input. Thus, the following construction

```
for i=1:10; plot(vec(i)); hold on; end
```

would not work. This limitation may be removed in a future version.

3.6.2 Example

Here is a function that takes two string arguments and returns the concatenation of them.

```
     strcattest.m
function strcattest(str1,str2)
  str3 = [str1,str2];
  printf('str1 = %s, str2 = %s, str3 = %s\n',str1,str2,str3);
```

We call `strcattest` using all three syntaxes.

```
--> strcattest('hi','ho')
str1 = hi, str2 = ho, str3 = hiho
--> strcattest 'hi' 'ho'
str1 = hi, str2 = ho, str3 = hiho
--> strcattest hi ho
str1 = hi, str2 = ho, str3 = hiho
```

3.7 VARARGIN Variable Input Arguments

3.7.1 Usage

FreeMat functions can take a variable number of input arguments by setting the last argument in the argument list to `varargin`. This special keyword indicates that all arguments to the function (beyond the last non-`varargin` keyword) are assigned to a cell array named `varargin` available to the function. Variable argument functions are usually used when writing driver functions, i.e., functions that need to pass arguments to another function. The general syntax for a function that takes a variable number of arguments is

```
  function [out_1,...,out_M] = fname(in_1,..,in_M,varargin)
```

Inside the function body, `varargin` collects the arguments to `fname` that are not assigned to the `in_k`.

3.7.2 Example

Here is a simple wrapper to `feval` that demonstrates the use of variable arguments functions.

```
     wrapcall.m
function wrapcall(fname,varargin)
  feval(fname,varargin{:});
```

Now we show a call of the `wrapcall` function with a number of arguments

```
--> wrapcall('printf','%f...%f\n',pi,e)
3.141593...2.718282
```

A more serious driver routine could, for example, optimize a one dimensional function that takes a number of auxilliary parameters that are passed through `varargin`.

3.8 VARARGOUT Variable Output Arguments

3.8.1 Usage

FreeMat functions can return a variable number of output arguments by setting the last argument in the argument list to `varargout`. This special keyword indicates that the number of return values is variable. The general syntax for a function that returns a variable number of outputs is

```
  function [out_1,...,out_M,varargout] = fname(in_1,...,in_M)
```

The function is responsible for ensuring that `varargout` is a cell array that contains the values to assign to the outputs beyond `out_M`. Generally, variable output functions use `nargout` to figure out how many outputs have been requested.

3.8.2 Example

This is a function that returns a varying number of values depending on the value of the argument.

```
     varoutfunc.m
function [varargout] = varoutfunc
  switch(nargout)
    case 1
      varargout = {'one of one'};
```

```
    case 2
      varargout = {'one of two','two of two'};
    case 3
      varargout = {'one of three','two of three','three of three'};
  end
```

Here are some examples of exercising `varoutfunc`:

```
--> [c1] = varoutfunc
c1 =
  <string>  - size: [1 10]
 one of one
--> [c1,c2] = varoutfunc
c1 =
  <string>  - size: [1 10]
 one of two
c2 =
  <string>  - size: [1 10]
 two of two
--> [c1,c2,c3] = varoutfunc
c1 =
  <string>  - size: [1 12]
 one of three
c2 =
  <string>  - size: [1 12]
 two of three
c3 =
  <string>  - size: [1 14]
 three of three
```

Chapter 4

Mathematical Operators

4.1 COLON Index Generation Operator

4.1.1 Usage

There are two distinct syntaxes for the colon : operator - the two argument form

```
y = a : c
```

and the three argument form

```
y = a : b : c
```

The two argument form is exactly equivalent to `a:1:c`. The output `y` is the vector

$$y = [a, a + b, a + 2b, \ldots, a + nb]$$

where `a+nb <= c`. There is a third form of the colon operator, the no-argument form used in indexing (see `indexing` for more details).

4.1.2 Function Internals

The colon operator turns out to be trickier to implement than one might believe at first, primarily because the floating point versions should do the right thing, which is not the obvious behavior. For example, suppose the user issues a three point colon command

```
y = a : b : c
```

The first question that one might need to answer is: how many points in this vector? If you answered

$$n = \frac{c - a}{b} + 1$$

then you would be doing the straighforward, but not correct thing. because a, b, and c are all floating point values, there are errors associated with each of the quantities that can lead to n not being an integer. A better way (and the way FreeMat currently does the calculation) is to compute the bounding values (for b positive)

$$n \in \left[\frac{(c-a) \rightarrow 0}{b \rightarrow \infty}, \frac{(c-a) \rightarrow \infty}{b \rightarrow 0} \right] + 1$$

where

$$x \rightarrow y$$

means we replace x by the floating point number that is closest to it in the direction of y. Once we have determined the number of points we have to compute the intermediate values

$$[a, a+b, a+2*b, \ldots, a+n*b]$$

but one can readily verify for themselves that this may *not* be the same as the vector

$$\mathrm{fliplr}[c, c-b, c-2*b, \ldots, c-n*b]$$

even for the case where

$$c = a + n*b$$

for some n. The reason is that the roundoff in the calculations may be different depending on the nature of the sum. FreeMat uses the following strategy to compute the double-colon vector:

1. The value `n` is computed by taking the floor of the larger value in the interval defined above.

2. If `n` falls inside the interval defined above, then it is assumed that the user intended `c = a + n*b`, and the symmetric algorithm is used. Otherwise, the nonsymmetric algorithm is used.

3. The symmetric algorithm computes the vector via

   $$[a, a+b, a+2b, \ldots, c-2b, c-b, c]$$

   working symmetrically from both ends of the vector (hence the nomenclature), while the nonsymmetric algorithm computes

   $$[a, a+b, a+2b, \ldots, a+nb]$$

   In practice, the entries are computed by repeated accumulation instead of multiplying the step size by an integer.

4. The real interval calculation is modified so that we get theexact same result with `a:b:c` and `c:-b:a` (which basically meansthat instead of moving towards infinity, we move towards the signed infinity where the sign is inherited from `b`).

If you think this is all very obscure, it is. But without it, you will be confronted by mysterious vectors where the last entry is dropped, or where the values show progressively larger amounts of accumulated roundoff error.

4.1.3 Examples

Some simple examples of index generation.

```
--> y = 1:4
y =
  <int32>  - size: [1 4]

Columns 1 to 4
 1  2  3  4
```

Now by half-steps:

```
--> y = 1:.5:4
y =
  <double>  - size: [1 7]

Columns 1 to 7
 1.0  1.5  2.0  2.5  3.0  3.5  4.0
```

Now going backwards (negative steps)

```
--> y = 4:-.5:1
y =
  <double>  - size: [1 7]

Columns 1 to 7
 4.0  3.5  3.0  2.5  2.0  1.5  1.0
```

If the endpoints are the same, one point is generated, regardless of the step size (middle argument)

```
--> y = 4:1:4
y =
  <int32>  - size: [1 1]
 4
```

If the endpoints define an empty interval, the output is an empty matrix:

```
--> y = 5:4
y =
  <int32>  - size: []
  []
```

4.2 COMPARISONOPS Array Comparison Operators

4.2.1 Usage

There are a total of six comparison operators available in FreeMat, all of which are binary operators with the following syntax

```
y = a < b
y = a <= b
y = a > b
y = a >= b
y = a ~= b
y = a == b
```

where `a` and `b` are numerical arrays or scalars, and `y` is a `logical` array of the appropriate size. Each of the operators has three modes of operation, summarized in the following list:

1. `a` is a scalar, `b` is an n-dimensional array - the output is then the same size as `b`, and contains the result of comparing each element in `b` to the scalar `a`.
2. `a` is an n-dimensional array, `b` is a scalar - the output is the same size as `a`, and contains the result of comparing each element in `a` to the scalar `b`.
3. `a` and `b` are both n-dimensional arrays of the same size - the output is then the same size as both `a` and `b`, and contains the result of an element-wise comparison between `a` and `b`.

The operators behave the same way as in `C`, with unequal types meing promoted using the standard type promotion rules prior to comparisons. The only difference is that in FreeMat, the not-equals operator is `~=` instead of `!=`.

4.2.2 Examples

Some simple examples of comparison operations. First a comparison with a scalar:

```
--> a = randn(1,5)
a =
  <double>  - size: [1 5]

Columns 1 to 3
 -0.939406097470965928  -0.006488070068068280  -0.053095354754894804

Columns 4 to 5
 -0.164794584325194143   0.010116755659839778
--> a>0
ans =
  <logical>  - size: [1 5]

Columns 1 to 5
 0  0  0  0  1
```

Next, we construct two vectors, and test for equality:

```
--> a = [1,2,5,7,3]
a =
  <int32>  - size: [1 5]
```

```
Columns 1 to 5
 1  2  5  7  3
--> b = [2,2,5,9,4]
b =
  <int32>  - size: [1 5]

Columns 1 to 5
 2  2  5  9  4
--> c = a == b
c =
  <logical>  - size: [1 5]

Columns 1 to 5
 0  1  1  0  0
```

4.3 DOTLEFTDIVIDE Element-wise Left-Division Operator

4.3.1 Usage

Divides two numerical arrays (elementwise) - gets its name from the fact that the divisor is on the left. There are two forms for its use, both with the same general syntax:

```
y = a .\ b
```

where `a` and `b` are `n`-dimensional arrays of numerical type. In the first case, the two arguments are the same size, in which case, the output `y` is the same size as the inputs, and is the element-wise division of `b` by `a`. In the second case, either `a` or `b` is a scalar, in which case `y` is the same size as the larger argument, and is the division of the scalar with each element of the other argument.

The type of `y` depends on the types of `a` and `b` using type promotion rules, with one important exception: unlike `C`, integer types are promoted to `double` prior to division.

4.3.2 Function Internals

There are three formulae for the dot-left-divide operator, depending on the sizes of the three arguments. In the most general case, in which the two arguments are the same size, the output is computed via:

$$y(m_1,\ldots,m_d) = \frac{b(m_1,\ldots,m_d)}{a(m_1,\ldots,m_d)}$$

If `a` is a scalar, then the output is computed via

$$y(m_1,\ldots,m_d) = \frac{b(m_1,\ldots,m_d)}{a}$$

On the other hand, if `b` is a scalar, then the output is computed via

$$y(m_1,\ldots,m_d) = \frac{b}{a(m_1,\ldots,m_d)}.$$

4.3.3 Examples

Here are some examples of using the dot-left-divide operator. First, a straight-forward usage of the .\\ operator. The first example is straightforward:

```
--> 3 .\ 8
ans =
  <double>  - size: [1 1]
 2.6666666666666665
```

Note that this is not the same as evaluating `8/3` in `C` - there, the output would be `2`, the result of the integer division.

We can also divide complex arguments:

```
--> a = 3 + 4*i
a =
  <complex>  - size: [1 1]
  3+ 4 i
--> b = 5 + 8*i
b =
  <complex>  - size: [1 1]
  5+ 8 i
--> c = b .\ a
c =
  <complex>  - size: [1 1]
  0.52808988-0.04494382 i
```

If a `complex` value is divided by a `double`, the result is promoted to `dcomplex`.

```
--> b = a .\ 2.0
b =
  <dcomplex>  - size: [1 1]
  0.24-0.32i
```

We can also demonstrate the three forms of the dot-left-divide operator. First the element-wise version:

```
--> a = [1,2;3,4]
a =
  <int32>  - size: [2 2]

Columns 1 to 2
 1  2
 3  4
--> b = [2,3;6,7]
b =
  <int32>  - size: [2 2]

Columns 1 to 2
```

```
 2  3
 6  7
--> c = a .\ b
c =
  <double>  - size: [2 2]

Columns 1 to 2
 2.00  1.50
 2.00  1.75
```

Then the scalar versions

```
--> c = a .\ 3
c =
  <double>  - size: [2 2]

Columns 1 to 2
 3.00  1.50
 1.00  0.75
--> c = 3 .\ a
c =
  <double>  - size: [2 2]

Columns 1 to 2
 0.3333333333333333  0.6666666666666666
 1.0000000000000000  1.3333333333333333
```

4.4 DOTPOWER Element-wise Power Operator

4.4.1 Usage

Raises one numerical array to another array (elementwise). There are three operators all with the same general syntax:

```
  y = a .^ b
```

The result `y` depends on which of the following three situations applies to the arguments `a` and `b`:

1. `a` is a scalar, `b` is an arbitrary `n`-dimensional numerical array, in which case the output is `a` raised to the power of each element of `b`, and the output is the same size as `b`.

2. `a` is an `n`-dimensional numerical array, and `b` is a scalar, then the output is the same size as `a`, and is defined by each element of `a` raised to the power `b`.

3. `a` and `b` are both `n`-dimensional numerical arrays of *the same size*. In this case, each element of the output is the corresponding element of `a` raised to the power defined by the corresponding element of `b`.

The output follows the standard type promotion rules, although types are not generally preserved under the power operation. In particular, integers are automatically converted to `double` type, and negative numbers raised to fractional powers can return complex values.

4.4.2 Function Internals

There are three formulae for this operator. For the first form

$$y(m_1,\ldots,m_d) = a^{b(m_1,\ldots,m_d)},$$

and the second form

$$y(m_1,\ldots,m_d) = a(m_1,\ldots,m_d)^b,$$

and in the third form

$$y(m_1,\ldots,m_d) = a(m_1,\ldots,m_d)^{b(m_1,\ldots,m_d)}.$$

4.4.3 Examples

We demonstrate the three forms of the dot-power operator using some simple examples. First, the case of a scalar raised to a series of values.

```
--> a = 2
a =
  <int32>  - size: [1 1]
 2
--> b = 1:4
b =
  <int32>  - size: [1 4]

Columns 1 to 4
 1  2  3  4
--> c = a.^b
c =
  <double>  - size: [1 4]

Columns 1 to 4
  2   4   8  16
```

The second case shows a vector raised to a scalar.

```
--> c = b.^a
c =
  <double>  - size: [1 4]

Columns 1 to 4
  1   4   9  16
```

The third case shows the most general use of the dot-power operator.

```
--> A = [1,2;3,2]
A =
  <int32>  - size: [2 2]

Columns 1 to 2
 1  2
 3  2
--> B = [2,1.5;0.5,0.6]
B =
  <double>  - size: [2 2]

Columns 1 to 2
 2.0  1.5
 0.5  0.6
--> C = A.^B
C =
  <double>  - size: [2 2]

Columns 1 to 2
 1.0000000000000000  2.8284271247461903
 1.7320508075688772  1.5157165665103980
```

4.5 DOTRIGHTDIVIDE Element-wise Right-Division Operator

4.5.1 Usage

Divides two numerical arrays (elementwise). There are two forms for its use, both with the same general syntax:

```
y = a ./ b
```

where a and b are n-dimensional arrays of numerical type. In the first case, the two arguments are the same size, in which case, the output y is the same size as the inputs, and is the element-wise division of b by a. In the second case, either a or b is a scalar, in which case y is the same size as the larger argument, and is the division of the scalar with each element of the other argument.

The type of y depends on the types of a and b using type promotion rules, with one important exception: unlike C, integer types are promoted to double prior to division.

4.5.2 Function Internals

There are three formulae for the dot-right-divide operator, depending on the sizes of the three arguments. In the most general case, in which the two arguments are the same size, the output is computed via:

$$y(m_1,\ldots,m_d) = \frac{a(m_1,\ldots,m_d)}{b(m_1,\ldots,m_d)}$$

If a is a scalar, then the output is computed via

$$y(m_1,\ldots,m_d) = \frac{a}{b(m_1,\ldots,m_d)}$$

On the other hand, if b is a scalar, then the output is computed via

$$y(m_1,\ldots,m_d) = \frac{a(m_1,\ldots,m_d)}{b}.$$

4.5.3 Examples

Here are some examples of using the dot-right-divide operator. First, a straight-forward usage of the ./ operator. The first example is straightforward:

```
--> 3 ./ 8
ans =
  <double>  - size: [1 1]
 0.375
```

Note that this is not the same as evaluating 3/8 in C - there, the output would be 0, the result of the integer division.

We can also divide complex arguments:

```
--> a = 3 + 4*i
a =
  <complex>  - size: [1 1]
  3+ 4 i
--> b = 5 + 8*i
b =
  <complex>  - size: [1 1]
  5+ 8 i
--> c = a ./ b
c =
  <complex>  - size: [1 1]
  0.52808988-0.04494382 i
```

If a complex value is divided by a double, the result is promoted to dcomplex.

```
--> b = a ./ 2.0
b =
  <dcomplex>  - size: [1 1]
 1.5+2.0i
```

We can also demonstrate the three forms of the dot-right-divide operator. First the element-wise version:

```
--> a = [1,2;3,4]
a =
  <int32>  - size: [2 2]
```

```
Columns 1 to 2
 1  2
 3  4
--> b = [2,3;6,7]
b =
  <int32>  - size: [2 2]

Columns 1 to 2
 2  3
 6  7
--> c = a ./ b
c =
  <double>  - size: [2 2]

Columns 1 to 2
 0.5000000000000000  0.6666666666666666
 0.5000000000000000  0.5714285714285714
```

Then the scalar versions

```
--> c = a ./ 3
c =
  <double>  - size: [2 2]

Columns 1 to 2
 0.3333333333333333  0.6666666666666666
 1.0000000000000000  1.3333333333333333
--> c = 3 ./ a
c =
  <double>  - size: [2 2]

Columns 1 to 2
 3.00  1.50
 1.00  0.75
```

4.6 DOTTIMES Element-wise Multiplication Operator

4.6.1 Usage

Multiplies two numerical arrays (elementwise). There are two forms for its use, both with the same general syntax:

```
  y = a .* b
```

where `a` and `b` are `n`-dimensional arrays of numerical type. In the first case, the two arguments are the same size, in which case, the output `y` is the same size as the inputs, and is the element-wise product of `a` and `b`. In the second case, either `a` or `b` is a scalar, in which case `y` is the same size as the larger argument, and is the product of the scalar with each element of the other argument.

The type of `y` depends on the types of `a` and `b` using type promotion rules. All of the types are preserved under multiplication except for integer types, which are promoted to `int32` prior to multiplication (same as `C`).

4.6.2 Function Internals

There are three formulae for the dot-times operator, depending on the sizes of the three arguments. In the most general case, in which the two arguments are the same size, the output is computed via:

$$y(m_1,\ldots,m_d) = a(m_1,\ldots,m_d) \times b(m_1,\ldots,m_d)$$

If `a` is a scalar, then the output is computed via

$$y(m_1,\ldots,m_d) = a \times b(m_1,\ldots,m_d).$$

On the other hand, if `b` is a scalar, then the output is computed via

$$y(m_1,\ldots,m_d) = a(m_1,\ldots,m_d) \times b.$$

4.6.3 Examples

Here are some examples of using the dottimes operator. First, a straight-forward usage of the `.*` operator. The first example is straightforward:

```
--> 3 .* 8
ans =
  <int32>  - size: [1 1]
 24
```

Note, however, that because of the way that input is parsed, eliminating the spaces `3.*8` results in the input being parsed as `3. * 8`, which yields a `double` result:

```
--> 3.*8
ans =
  <int32>  - size: [1 1]
 24
```

This is really an invokation of the `times` operator.

Next, we use the floating point syntax to force one of the arguments to be a `double`, which results in the output being `double`:

```
--> 3.1 .* 2
ans =
  <double>  - size: [1 1]
 6.2
```

Note that if one of the arguments is complex-valued, the output will be complex also.

```
--> a = 3 + 4*i
a =
  <complex>  - size: [1 1]
  3+ 4 i
--> b = a .* 2.0f
b =
  <complex>  - size: [1 1]
  6+ 8 i
```

If a complex value is multiplied by a double, the result is promoted to dcomplex.

```
--> b = a .* 2.0
b =
  <dcomplex>  - size: [1 1]
  6+ 8i
```

We can also demonstrate the three forms of the dottimes operator. First the element-wise version:

```
--> a = [1,2;3,4]
a =
  <int32>  - size: [2 2]

Columns 1 to 2
 1  2
 3  4
--> b = [2,3;6,7]
b =
  <int32>  - size: [2 2]

Columns 1 to 2
 2  3
 6  7
--> c = a .* b
c =
  <int32>  - size: [2 2]

Columns 1 to 2
  2   6
 18  28
```

Then the scalar versions

```
--> c = a .* 3
c =
  <int32>  - size: [2 2]

Columns 1 to 2
  3   6
```

```
  9  12
--> c = 3 .* a
c =
  <int32>  - size: [2 2]

Columns 1 to 2
  3   6
  9  12
```

4.7 HERMITIAN Matrix Hermitian (Conjugate Transpose) Operator

4.7.1 Usage

Computes the Hermitian of the argument (a 2D matrix). The syntax for its use is

```
  y = a';
```

where `a` is a `M x N` numerical matrix. The output `y` is a numerical matrix of the same type of size `N x M`. This operator is the conjugating transpose, which is different from the transpose operator `.'` (which does not conjugate complex values).

4.7.2 Function Internals

The Hermitian operator is defined simply as

$$y_{i,j} = \overline{a_{j,i}}$$

where `y_ij` is the element in the `i`th row and `j`th column of the output matrix `y`.

4.7.3 Examples

A simple transpose example:

```
--> A = [1,2,0;4,1,-1]
A =
  <int32>  - size: [2 3]

Columns 1 to 3
  1   2   0
  4   1  -1
--> A'
ans =
  <int32>  - size: [3 2]

Columns 1 to 2
  1   4
```

```
  2   1
  0  -1
```

Here, we use a complex matrix to demonstrate how the Hermitian operator conjugates the entries.

```
--> A = [1+i,2-i]
A =
  <complex>  - size: [1 2]

Columns 1 to 2
   1+  1 i    2 -1 i
--> A.'
ans =
  <complex>  - size: [2 1]

Columns 1 to 1
   1+  1 i
   2 -1 i
```

4.8 LEFTDIVIDE Matrix Equation Solver/Divide Operator

4.8.1 Usage

The divide operator `\` is really a combination of three operators, all of which have the same general syntax:

```
  Y = A \ B
```

where `A` and `B` are arrays of numerical type. The result `Y` depends on which of the following three situations applies to the arguments `A` and `B`:

1. `A` is a scalar, `B` is an arbitrary `n`-dimensional numerical array, in which case the output is each element of `B` divided by the scalar `A`.

2. `B` is a scalar, `A` is an arbitrary `n`-dimensional numerical array, in which case the output is the scalar `B` divided by each element of `A`.

3. `A,B` are matrices with the same number of rows, i.e., `A` is of size `M x K`, and `B` is of size `M x L`, in which case the output is of size `K x L`.

The output follows the standard type promotion rules, although in the first two cases, if `A` and `B` are integers, the output is an integer also, while in the third case if `A` and `B` are integers, the output is of type `double`.

A few additional words about the third version, in which `A` and `B` are matrices. Very loosely speaking, `Y` is the matrix that satisfies `A * Y = B`. In cases where such a matrix exists. If such a matrix does not exist, then a matrix `Y` is returned that approximates `A * Y \approx B`.

4.8.2 Function Internals

There are three formulae for the times operator. For the first form

$$Y(m_1,\ldots,m_d) = \frac{B(m_1,\ldots,m_d)}{A},$$

and the second form

$$Y(m_1,\ldots,m_d) = \frac{B}{A(m_1,\ldots,m_d)}.$$

In the third form, the calculation of the output depends on the size of `A`. Because each column of `B` is treated independantly, we can rewrite the equation `A Y = B` as

$$A[y_1, y_2, \ldots, y_l] = [b_1, b_2, \ldots, b_l]$$

where `y_i` are the columns of `Y`, and `b_i` are the columns of the matrix `B`. If `A` is a square matrix, then the LAPACK routine `*gesvx` (where the `*` is replaced with `sdcz` depending on the type of the arguments) is used, which uses an LU decomposition of `A` to solve the sequence of equations sequentially. If `A` is singular, then a warning is emitted.

On the other hand, if `A` is rectangular, then the LAPACK routine `*gelsy` is used. Note that these routines are designed to work with matrices `A` that are full rank - either full column rank or full row rank. If `A` fails to satisfy this assumption, a warning is emitted. If `A` has full column rank (and thus necessarily has more rows than columns), then theoretically, this operator finds the columns `y_i` that satisfy:

$$y_i = \arg\min_y \|Ay - b_i\|_2$$

and each column is thus the Least Squares solution of `A y = b_i`. On the other hand, if `A` has full row rank (and thus necessarily has more columns than rows), then theoretically, this operator finds the columns `y_i` that satisfy

$$y_i = \arg\min_{Ay=b_i} \|y\|_2$$

and each column is thus the Minimum Norm vector `y_i` that satisfies `A y_i = b_i`. In the event that the matrix `A` is neither full row rank nor full column rank, a solution is returned, that is the minimum norm least squares solution. The solution is computed using an orthogonal factorization technique that is documented in the LAPACK User's Guide (see the References section for details).

4.8.3 Examples

Here are some simple examples of the divide operator. We start with a simple example of a full rank, square matrix:

```
--> A = [1,1;0,1]
A =
  <int32>  - size: [2 2]

Columns 1 to 2
 1  1
 0  1
```

Suppose we wish to solve

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

(which by inspection has the solution `y_1 = 1`, `y_2 = 2`). Thus we compute:

```
--> B = [3;2]
B =
  <int32>  - size: [2 1]

Columns 1 to 1
 3
 2
--> Y = A\B
Y =
  <double>  - size: [2 1]

Columns 1 to 1
 1
 2
```

Suppose we wish to solve a trivial Least Squares (LS) problem. We want to find a simple scaling of the vector `[1;1]` that is closest to the point `[2,1]`. This is equivalent to solving

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} y = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

in a least squares sense. For fun, we can calculate the solution using calculus by hand. The error we wish to minimize is

$$\varepsilon(y) = (y-2)^2 + (y-1)^2.$$

Taking a derivative with respect to `y`, and setting to zero (which we must have for an extrema when `y` is unconstrained)

$$2(y-2) + 2(y-1) = 0$$

which we can simplify to `4y = 6` or `y = 3/2` (we must, technically, check to make sure this is a minimum, and not a maximum or an inflection point). Here is the same calculation performed using FreeMat:

```
--> A = [1;1]
A =
  <int32>  - size: [2 1]

Columns 1 to 1
 1
 1
--> B = [2;1]
B =
  <int32>  - size: [2 1]
```

```
Columns 1 to 1
 2
 1
--> A\B
ans =
  <double>  - size: [1 1]
 1.4999999999999998
```

which is the same solution.

4.9 LOGICALOPS Logical Array Operators

4.9.1 Usage

There are three Boolean operators available in FreeMat. The syntax for their use is:

```
y = ~x
y = a & b
y = a | b
```

where `x`, `a` and `b` are `logical` arrays. The operators are

- NOT (~) - output `y` is true if the corresponding element of `x` is false, and ouput `y` is false if the corresponding element of `x` is true.
- OR (—) - output `y` is true if corresponding element of `a` is true or if corresponding element of `b` is true (or if both are true).
- AND (\&) - output `y` is true only if both the corresponding elements of `a` and `b` are both true.

The binary operators AND and OR can take scalar arguments as well as vector arguments, in which case, the scalar is operated on with each element of the vector. As of version 1.10, FreeMat supports `shortcut` evaluation. This means that if we have two expressions

```
if (expr1 & expr2)
```

then if `expr1` evaluates to `false`, then `expr2` is not evaluated at all. Similarly, for the expression

```
if (expr1 | expr2)
```

then if `expr1` evaluates to `true`, then `expr2` is not evaluated at all. Shortcut evaluation is useful for doing a sequence of tests, each of which is not valid unless the prior test is successful. For example,

```
if isa(p,'string') & strcmp(p,'fro')
```

is not valid without shortcut evaluation (if `p` is an integer, for example, the first test returns false, and an attempt to evaluate the second expression would lead to an error). Note that shortcut evaluation only works with scalar expressions.

4.9.2 Examples

Some simple examples of logical operators. Suppose we want to calculate the exclusive-or (XOR) of two vectors of logical variables. First, we create a pair of vectors to perform the XOR operation on:

```
--> a = (randn(1,6)>0)
a =
  <logical>  - size: [1 6]

Columns 1 to 6
 0  0  1  0  1  0
--> b = (randn(1,6)>0)
b =
  <logical>  - size: [1 6]

Columns 1 to 6
 0  1  0  0  0  1
```

Next, we can compute the OR of `a` and `b`:

```
--> c = a | b
c =
  <logical>  - size: [1 6]

Columns 1 to 6
 0  1  1  0  1  1
```

However, the XOR and OR operations differ on the fifth entry - the XOR would be false, since it is true if and only if exactly one of the two inputs is true. To isolate this case, we can AND the two vectors, to find exactly those entries that appear as true in both `a` and `b`:

```
--> d = a & b
d =
  <logical>  - size: [1 6]

Columns 1 to 6
 0  0  0  0  0  0
```

At this point, we can modify the contents of `c` in two ways – the Boolean way is to AND `\sim d` with `c`, like so

```
--> xor = c & (~d)
xor =
  <logical>  - size: [1 6]

Columns 1 to 6
 0  1  1  0  1  1
```

The other way to do this is simply force `c(d) = 0`, which uses the logical indexing mode of FreeMat (see the chapter on indexing for more details). This, however, will cause `c` to become an `int32` type, as opposed to a logical type.

```
--> c(d) = 0
c =
  <logical>  - size: [1 6]

Columns 1 to 6
 0  1  1  0  1  1
```

4.10 MINUS Subtraction Operator

4.10.1 Usage

Subtracts two numerical arrays (elementwise). There are two forms for its use, both with the same general syntax:

```
  y = a - b
```

where `a` and `b` are `n`-dimensional arrays of numerical type. In the first case, the two arguments are the same size, in which case, the output `y` is the same size as the inputs, and is the element-wise difference of `a` and `b`. In the second case, either `a` or `b` is a scalar, in which case `y` is the same size as the larger argument, and is the difference of the scalar to each element of the other argument.

The type of `y` depends on the types of `a` and `b` using the type promotion rules. The types are ordered as:

1. `uint8` - unsigned, 8-bit integers range `[0,255]`
2. `int8` - signed, 8-bit integers `[-127,128]`
3. `uint16` - unsigned, 16-bit integers `[0,65535]`
4. `int16` - signed, 16-bit integers `[-32768,32767]`
5. `uint32` - unsigned, 32-bit integers `[0,4294967295]`
6. `int32` - signed, 32-bit integers `[-2147483648,2147483647]`
7. `float` - 32-bit floating point
8. `double` - 64-bit floating point
9. `complex` - 32-bit complex floating point
10. `dcomplex` - 64-bit complex floating point

Note that the type promotion and combination rules work similar to `C`. Numerical overflow rules are also the same as `C`.

4.10.2 Function Internals

There are three formulae for the subtraction operator, depending on the sizes of the three arguments. In the most general case, in which the two arguments are the same size, the output is computed via:

$$y(m_1,\ldots,m_d) = a(m_1,\ldots,m_d) - b(m_1,\ldots,m_d)$$

If `a` is a scalar, then the output is computed via

$$y(m_1,\ldots,m_d) = a - b(m_1,\ldots,m_d).$$

On the other hand, if `b` is a scalar, then the output is computed via

$$y(m_1,\ldots,m_d) = a(m_1,\ldots,m_d) - b.$$

4.10.3 Examples

Here are some examples of using the subtraction operator. First, a straight-forward usage of the minus operator. The first example is straightforward - the `int32` is the default type used for integer constants (same as in `C`), hence the output is the same type:

```
--> 3 - 8
ans =
  <int32>  - size: [1 1]
 -5
```

Next, we use the floating point syntax to force one of the arguments to be a `double`, which results in the output being `double`:

```
--> 3.1 - 2
ans =
  <double>  - size: [1 1]
 1.1
```

Note that if one of the arguments is complex-valued, the output will be complex also.

```
--> a = 3 + 4*i
a =
  <complex>  - size: [1 1]
  3+ 4 i
--> b = a - 2.0f
b =
  <complex>  - size: [1 1]
  1+ 4 i
```

If a `double` value is subtracted from a `complex`, the result is promoted to `dcomplex`.

```
--> b = a - 2.0
b =
  <dcomplex>  - size: [1 1]
  1+ 4i
```

We can also demonstrate the three forms of the subtraction operator. First the element-wise version:

```
--> a = [1,2;3,4]
a =
  <int32>  - size: [2 2]

Columns 1 to 2
 1  2
 3  4
--> b = [2,3;6,7]
b =
  <int32>  - size: [2 2]

Columns 1 to 2
 2  3
 6  7
--> c = a - b
c =
  <int32>  - size: [2 2]

Columns 1 to 2
 -1  -1
 -3  -3
```

Then the scalar versions

```
--> c = a - 1
c =
  <int32>  - size: [2 2]

Columns 1 to 2
 0  1
 2  3
--> c = 1 - b
c =
  <int32>  - size: [2 2]

Columns 1 to 2
 -1  -2
 -5  -6
```

4.11 PLUS Addition Operator

4.11.1 Usage

Adds two numerical arrays (elementwise) together. There are two forms for its use, both with the same general syntax:

```
y = a + b
```

where `a` and `b` are `n`-dimensional arrays of numerical type. In the first case, the two arguments are the same size, in which case, the output `y` is the same size as the inputs, and is the element-wise the sum of `a` and `b`. In the second case, either `a` or `b` is a scalar, in which case `y` is the same size as the larger argument, and is the sum of the scalar added to each element of the other argument.

The type of `y` depends on the types of `a` and `b` using the type promotion rules. The types are ordered as:

1. `uint8` - unsigned, 8-bit integers range `[0,255]`
2. `int8` - signed, 8-bit integers `[-127,128]`
3. `uint16` - unsigned, 16-bit integers `[0,65535]`
4. `int16` - signed, 16-bit integers `[-32768,32767]`
5. `uint32` - unsigned, 32-bit integers `[0,4294967295]`
6. `int32` - signed, 32-bit integers `[-2147483648,2147483647]`
7. `float` - 32-bit floating point
8. `double` - 64-bit floating point
9. `complex` - 32-bit complex floating point
10. `dcomplex` - 64-bit complex floating point

Note that the type promotion and combination rules work similar to `C`. Numerical overflow rules are also the same as `C`.

4.11.2 Function Internals

There are three formulae for the addition operator, depending on the sizes of the three arguments. In the most general case, in which the two arguments are the same size, the output is computed via:

$$y(m_1,\ldots,m_d) = a(m_1,\ldots,m_d) + b(m_1,\ldots,m_d)$$

If `a` is a scalar, then the output is computed via

$$y(m_1,\ldots,m_d) = a + b(m_1,\ldots,m_d).$$

On the other hand, if `b` is a scalar, then the output is computed via

$$y(m_1,\ldots,m_d) = a(m_1,\ldots,m_d) + b.$$

4.11.3 Examples

Here are some examples of using the addition operator. First, a straight-forward usage of the plus operator. The first example is straightforward - the `int32` is the default type used for integer constants (same as in `C`), hence the output is the same type:

```
--> 3 + 8
ans =
  <int32>  - size: [1 1]
 11
```

Next, we use the floating point syntax to force one of the arguments to be a `double`, which results in the output being `double`:

```
--> 3.1 + 2
ans =
  <double>  - size: [1 1]
 5.1
```

Note that if one of the arguments is complex-valued, the output will be complex also.

```
--> a = 3 + 4*i
a =
  <complex>  - size: [1 1]
  3+ 4 i
--> b = a + 2.0f
b =
  <complex>  - size: [1 1]
  5+ 4 i
```

If a `complex` value is added to a `double`, the result is promoted to `dcomplex`.

```
--> b = a + 2.0
b =
  <dcomplex>  - size: [1 1]
  5+ 4i
```

We can also demonstrate the three forms of the addition operator. First the element-wise version:

```
--> a = [1,2;3,4]
a =
  <int32>  - size: [2 2]

Columns 1 to 2
 1  2
 3  4
--> b = [2,3;6,7]
b =
  <int32>  - size: [2 2]
```

```
Columns 1 to 2
 2  3
 6  7
--> c = a + b
c =
  <int32>  - size: [2 2]

Columns 1 to 2
  3   5
  9  11
```

Then the scalar versions

```
--> c = a + 1
c =
  <int32>  - size: [2 2]

Columns 1 to 2
 2  3
 4  5
--> c = 1 + b
c =
  <int32>  - size: [2 2]

Columns 1 to 2
 3  4
 7  8
```

4.12 POWER Matrix Power Operator

4.12.1 Usage

The power operator for scalars and square matrices. This operator is really a combination of two operators, both of which have the same general syntax:

```
  y = a ^ b
```

The exact action taken by this operator, and the size and type of the output, depends on which of the two configurations of `a` and `b` is present:

1. `a` is a scalar, `b` is a square matrix

2. `a` is a square matrix, `b` is a scalar

4.12.2 Function Internals

In the first case that `a` is a scalar, and `b` is a square matrix, the matrix power is defined in terms of the eigenvalue decomposition of `b`. Let `b` have the following eigen-decomposition (problems arise with non-symmetric matrices `b`, so let us assume that `b` is symmetric):

$$b = E \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & \lambda_n \end{bmatrix} E^{-1}$$

Then `a` raised to the power `b` is defined as

$$a^b = E \begin{bmatrix} a^{\lambda_1} & 0 & \cdots & 0 \\ 0 & a^{\lambda_2} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & a^{\lambda_n} \end{bmatrix} E^{-1}$$

Similarly, if `a` is a square matrix, then `a` has the following eigen-decomposition (again, suppose `a` is symmetric):

$$a = E \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & \lambda_n \end{bmatrix} E^{-1}$$

Then `a` raised to the power `b` is defined as

$$a^b = E \begin{bmatrix} \lambda_1^b & 0 & \cdots & 0 \\ 0 & \lambda_2^b & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & \lambda_n^b \end{bmatrix} E^{-1}$$

4.12.3 Examples

We first define a simple `2 x 2` symmetric matrix

```
--> A = 1.5
A =
  <double>  - size: [1 1]
 1.5
--> B = [1,.2;.2,1]
B =
  <double>  - size: [2 2]
```

```
Columns 1 to 2
 1.0  0.2
 0.2  1.0
```

First, we raise B to the (scalar power) A:

```
--> C = B^A
C =
  <double>  - size: [2 2]

Columns 1 to 2
 1.0150379454061658  0.2994961926062329
 0.2994961926062330  1.0150379454061658
```

Next, we raise A to the matrix power B:

```
--> C = A^B
C =
  <double>  - size: [2 2]

Columns 1 to 2
 1.50493476200956966  0.12177289478697813
 0.12177289478697809  1.50493476200956966
```

4.13 RIGHTDIVIDE Matrix Equation Solver/Divide Operator

4.13.1 Usage

The divide operator `/` is really a combination of three operators, all of which have the same general syntax:

```
  Y = A / B
```

where `A` and `B` are arrays of numerical type. The result `Y` depends on which of the following three situations applies to the arguments `A` and `B`:

1. `A` is a scalar, `B` is an arbitrary `n`-dimensional numerical array, in which case the output is the scalar `A` divided into each element of `B`.

2. `B` is a scalar, `A` is an arbitrary `n`-dimensional numerical array, in which case the output is each element of `A` divided by the scalar `B`.

3. `A,B` are matrices with the same number of columns, i.e., `A` is of size `K x M`, and `B` is of size `L x M`, in which case the output is of size `K x L`.

The output follows the standard type promotion rules, although in the first two cases, if `A` and `B` are integers, the output is an integer also, while in the third case if `A` and `B` are integers, the output is of type `double`.

4.13.2 Function Internals

There are three formulae for the times operator. For the first form

$$Y(m_1,\ldots,m_d) = \frac{A}{B(m_1,\ldots,m_d)},$$

and the second form

$$Y(m_1,\ldots,m_d) = \frac{A(m_1,\ldots,m_d)}{B}.$$

In the third form, the output is defined as:

$$Y = (B'\backslash A')'$$

and is used in the equation `Y B = A`.

4.13.3 Examples

The right-divide operator is much less frequently used than the left-divide operator, but the concepts are similar. It can be used to find least-squares and minimum norm solutions. It can also be used to solve systems of equations in much the same way. Here's a simple example:

```
--> B = [1,1;0,1];
--> A = [4,5]
A =
  <int32>  - size: [1 2]

Columns 1 to 2
 4  5
--> A/B
ans =
  <double>  - size: [1 2]

Columns 1 to 2
 4  1
```

4.14 TIMES Matrix Multiply Operator

4.14.1 Usage

Multiplies two numerical arrays. This operator is really a combination of three operators, all of which have the same general syntax:

```
  y = a * b
```

where `a` and `b` are arrays of numerical type. The result `y` depends on which of the following three situations applies to the arguments `a` and `b`:

1. `a` is a scalar, `b` is an arbitrary `n`-dimensional numerical array, in which case the output is the element-wise product of `b` with the scalar `a`.

2. `b` is a scalar, `a` is an arbitrary `n`-dimensional numerical array, in which case the output is the element-wise product of `a` with the scalar `b`.

3. `a,b` are conformant matrices, i.e., `a` is of size `M x K`, and `b` is of size `K x N`, in which case the output is of size `M x N` and is the matrix product of `a`, and `b`.

The output follows the standard type promotion rules, although in the first two cases, if `a` and `b` are integers, the output is an integer also, while in the third case if `a` and `b` are integers, ,the output is of type `double`.

4.14.2 Function Internals

There are three formulae for the times operator. For the first form

$$y(m_1,\ldots,m_d) = a \times b(m_1,\ldots,m_d),$$

and the second form

$$y(m_1,\ldots,m_d) = a(m_1,\ldots,m_d) \times b.$$

In the third form, the output is the matrix product of the arguments

$$y(m,n) = \sum_{k=1}^{K} a(m,k)b(k,n)$$

4.14.3 Examples

Here are some examples of using the matrix multiplication operator. First, the scalar examples (types 1 and 2 from the list above):

```
--> a = [1,3,4;0,2,1]
a =
  <int32>  - size: [2 3]

Columns 1 to 3
 1  3  4
 0  2  1
--> b = a * 2
b =
  <int32>  - size: [2 3]

Columns 1 to 3
 2  6  8
 0  4  2
```

The matrix form, where the first argument is `2 x 3`, and the second argument is `3 x 1`, so that the product is size `2 x 1`.

```
--> a = [1,2,0;4,2,3]
a =
  <int32>  - size: [2 3]

Columns 1 to 3
 1  2  0
 4  2  3
--> b = [5;3;1]
b =
  <int32>  - size: [3 1]

Columns 1 to 1
 5
 3
 1
--> c = a*b
c =
  <double>  - size: [2 1]

Columns 1 to 1
 11
 29
```

Note that the output is double precision.

4.15 TRANSPOSE Matrix Transpose Operator

4.15.1 Usage

Performs a transpose of the argument (a 2D matrix). The syntax for its use is

```
  y = a.';
```

where `a` is a `M x N` numerical matrix. The output `y` is a numerical matrix of the same type of size `N x M`. This operator is the non-conjugating transpose, which is different from the Hermitian operator `'` (which conjugates complex values).

4.15.2 Function Internals

The transpose operator is defined simply as

$$y_{i,j} = a_{j,i}$$

where `y_ij` is the element in the `i`th row and `j`th column of the output matrix `y`.

4.15.3 Examples

A simple transpose example:

```
--> A = [1,2,0;4,1,-1]
A =
  <int32>  - size: [2 3]

Columns 1 to 3
  1   2   0
  4   1  -1
--> A.'
ans =
  <int32>  - size: [3 2]

Columns 1 to 2
  1   4
  2   1
  0  -1
```

Here, we use a complex matrix to demonstrate how the transpose does *not* conjugate the entries.

```
--> A = [1+i,2-i]
A =
  <complex>  - size: [1 2]

Columns 1 to 2
   1+  1 i    2 -1 i
--> A.'
ans =
  <complex>  - size: [2 1]

Columns 1 to 1
   1+  1 i
   2 -1 i
```

Chapter 5

Flow Control

5.1 BREAK Exit Execution In Loop

5.1.1 Usage

The `break` statement is used to exit a loop prematurely. It can be used inside a `for` loop or a `while` loop. The syntax for its use is

```
break
```

inside the body of the loop. The `break` statement forces execution to exit the loop immediately.

5.1.2 Example

Here is a simple example of how `break` exits the loop. We have a loop that sums integers from `1` to `10`, but that stops prematurely at `5` using a `break`. We will use a `while` loop.

```
     break_ex.m
function accum = break_ex
  accum = 0;
  i = 1;
  while (i<=10)
    accum = accum + i;
    if (i == 5)
      break;
    end
    i = i + 1;
  end
```

The function is exercised here:

```
--> break_ex
ans =
  <int32>  - size: [1 1]
```

```
 15
--> sum(1:5)
ans =
  <int32>  - size: [1 1]
 15
```

5.2 CONTINUE Continue Execution In Loop

5.2.1 Usage

The `continue` statement is used to change the order of execution within a loop. The `continue` statement can be used inside a `for` loop or a `while` loop. The syntax for its use is

```
   continue
```

inside the body of the loop. The `continue` statement forces execution to start at the top of the loop with the next iteration. The examples section shows how the `continue` statement works.

5.2.2 Example

Here is a simple example of using a `continue` statement. We want to sum the integers from `1` to `10`, but not the number `5`. We will use a `for` loop and a continue statement.

```
     continue_ex.m
function accum = continue_ex
  accum = 0;
  for i=1:10
    if (i==5)
      continue;
    end
    accum = accum + 1; %skipped if i == 5!
  end
```

The function is exercised here:

```
--> continue_ex
ans =
  <int32>  - size: [1 1]
 9
--> sum([1:4,6:10])
ans =
  <int32>  - size: [1 1]
 50
```

5.3 ERROR Causes an Error Condition Raised

5.3.1 Usage

The `error` function causes an error condition (exception to be raised). The general syntax for its use is

```
error(s),
```

where `s` is the string message describing the error. The `error` function is usually used in conjunction with `try` and `catch` to provide error handling.

5.3.2 Example

Here is a simple example of an `error` being issued by a function `evenoddtest`:

```
     evenoddtest.m
function evenoddtest(n)
  if (n==0)
    error('zero is neither even nor odd');
  elseif (~isa(n,'int32'))
    error('expecting integer argument');
  end;
  if (n==int32(n/2)*2)
    printf('%d is even\n',n);
  else
    printf('%d is odd\n',n);
  end
```

The normal command line prompt `-->` simply prints the error that occured.

```
--> evenoddtest(4)
4 is even
--> evenoddtest(5)
5 is odd
--> evenoddtest(0)
Error: zero is neither even nor odd
In base(base), line 0, column 0
In Eval(evenoddtest(0)), line 1, column 12
In evenoddtest(evenoddtest), line 3, column 10
[evenoddtest,3] D-> evenoddtest(pi)
Error: expecting integer argument
In base(base), line 0, column 0
In Eval(evenoddtest(0)), line 1, column 12
In evenoddtest(evenoddtest), line 3, column 10
In Eval(evenoddtest(pi)), line 1, column 12
In evenoddtest(evenoddtest), line 5, column 10
```

5.4 FOR For Loop

5.4.1 Usage

The `for` loop executes a set of statements with an index variable looping through each element in a vector. The syntax of a `for` loop is one of the following:

```
  for (variable=expression)
     statements
  end
```

Alternately, the parenthesis can be eliminated

```
  for variable=expression
     statements
  end
```

or alternately, the index variable can be pre-initialized with the vector of values it is going to take:

```
  for variable
     statements
  end
```

The third form is essentially equivalent to `for variable=variable`, where `variable` is both the index variable and the set of values over which the for loop executes. See the examples section for an example of this form of the `for` loop.

5.4.2 Examples

Here we write `for` loops to add all the integers from `1` to `100`. We will use all three forms of the `for` statement.

```
--> accum = 0;
--> for (i=1:100); accum = accum + i; end
--> accum
ans =
  <int32>  - size: [1 1]
 5050
```

The second form is functionally the same, without the extra parenthesis

```
--> accum = 0;
--> for i=1:100; accum = accum + i; end
--> accum
ans =
  <int32>  - size: [1 1]
 5050
```

In the third example, we pre-initialize the loop variable with the values it is to take

5.5 IF-ELSEIF-ELSE Conditional Statements

5.5.1 Usage

The `if` and `else` statements form a control structure for conditional execution. The general syntax involves an `if` test, followed by zero or more `elseif` clauses, and finally an optional `else` clause:

```
  if conditional_expression_1
    statements_1
  elseif conditional_expression_2
    statements_2
  elseif conditional_expresiion_3
    statements_3
  ...
  else
    statements_N
  end
```

Note that a conditional expression is considered true if the real part of the result of the expression contains any non-zero elements (this strange convention is adopted for compatibility with MATLAB).

5.5.2 Examples

Here is an example of a function that uses an `if` statement

```
     if_test.m
function c = if_test(a)
  if (a == 1)
     c = 'one';
  elseif (a==2)
     c = 'two';
  elseif (a==3)
     c = 'three';
  else
     c = 'something else';
  end
```

Some examples of `if_test` in action:

```
--> if_test(1)
ans =
  <string>  - size: [1 3]
 one
--> if_test(2)
ans =
  <string>  - size: [1 3]
 two
--> if_test(3)
```

```
ans =
  <string>  - size: [1 5]
 three
--> if_test(pi)
ans =
  <string>  - size: [1 14]
 something else
```

5.6 KEYBOARD Initiate Interactive Debug Session

5.6.1 Usage

The keyboard statement is used to initiate an interactive session at a specific point in a function. The general syntax for the keyboard statement is

```
keyboard
```

A keyboard statement can be issued in a script, in a function, or from within another keyboard session. The result of a keyboard statement is that execution of the program is halted, and you are given a prompt of the form:

```
[scope,n] -->
```

where scope is the current scope of execution (either the name of the function we are executing, or base otherwise). And n is the depth of the keyboard session. If, for example, we are in a keyboard session, and we call a function that issues another keyboard session, the depth of that second session will be one higher. Put another way, n is the number of return statements you have to issue to get back to the base workspace. Incidentally, a return is how you exit the keyboard session and resume execution of the program from where it left off. A retall can be used to shortcut execution and return to the base workspace.

The keyboard statement is an excellent tool for debugging FreeMat code, and along with eval provide a unique set of capabilities not usually found in compiled environments. Indeed, the keyboard statement is equivalent to a debugger breakpoint in more traditional environments, but with significantly more inspection power.

5.6.2 Example

Here we demonstrate a two-level keyboard situation. We have a simple function that calls keyboard internally:

```
     key_one.m
function c = key_one(a,b)
c = a + b;
keyboard
```

Now, we execute the function from the base workspace, and at the keyboard prompt, we call it again. This action puts us at depth 2. We can confirm that we are in the second invocation of the function by examining the arguments. We then issue two return statements to return to the base workspace.

```
--> key_one(1,2)
[key_one,3] --> key_one(5,7)
[key_one,3] --> a
ans =
  <int32>  - size: [1 1]
 5
[key_one,3] --> b
ans =
  <int32>  - size: [1 1]
 7
[key_one,3] --> c
ans =
  <int32>  - size: [1 1]
 12
[key_one,3] --> return
ans =
  <int32>  - size: [1 1]
 12
[key_one,3] --> a
ans =
  <int32>  - size: [1 1]
 1
[key_one,3] --> b
ans =
  <int32>  - size: [1 1]
 2
[key_one,3] --> c
ans =
  <int32>  - size: [1 1]
 3
[key_one,3] --> return
ans =
  <int32>  - size: [1 1]
 3
```

5.7 LASTERR Retrieve Last Error Message

5.7.1 Usage

Either returns or sets the last error message. The general syntax for its use is either

```
msg = lasterr
```

which returns the last error message that occured, or

```
lasterr(msg)
```

which sets the contents of the last error message.

5.7.2 Example

Here is an example of using the `error` function to set the last error, and then retrieving it using lasterr.

```
--> try; error('Test error message'); catch; end;
lasterr
ans =
  <string>  - size: [1 18]
 Test error message
```

Or equivalently, using the second form:

```
--> lasterr('Test message');
--> lasterr
ans =
  <string>  - size: [1 12]
 Test message
```

5.8 RETALL Return From All Keyboard Sessions

5.8.1 Usage

The `retall` statement is used to return to the base workspace from a nested `keyboard` session. It is equivalent to forcing execution to return to the main prompt, regardless of the level of nesting of `keyboard` sessions, or which functions are running. The syntax is simple

```
   retall
```

The `retall` is a convenient way to stop debugging. In the process of debugging a complex program or set of functions, you may find yourself 5 function calls down into the program only to discover the problem. After fixing it, issueing a `retall` effectively forces FreeMat to exit your program and return to the interactive prompt.

5.8.2 Example

Here we demonstrate an extreme example of `retall`. We are debugging a recursive function `self` to calculate the sum of the first N integers. When the function is called, a `keyboard` session is initiated after the function has called itself N times. At this `keyboard` prompt, we issue another call to `self` and get another `keyboard` prompt, this time with a depth of 2. A `retall` statement returns us to the top level without executing the remainder of either the first or second call to `self`:

```
     self.m
function y = self(n)
  if (n>1)
    y = n + self(n-1);
    printf('y is %d\n',y);
  else
```

```
    y = 1;
    printf('y is initialized to one\n');
    keyboard
  end
```

```
--> self(4)
y is initialized to one
[self,8] --> self(6)
y is initialized to one
[self,8] --> retall
```

5.9 RETURN Return From Function

5.9.1 Usage

The `return` statement is used to immediately return from a function, or to return from a `keyboard` session. The syntax for its use is

```
  return
```

Inside a function, a `return` statement causes FreeMat to exit the function immediately. When a `keyboard` session is active, the `return` statement causes execution to resume where the `keyboard` session started.

5.9.2 Example

In the first example, we define a function that uses a `return` to exit the function if a certain test condition is satisfied.

```
     return_func.m
function ret = return_func(a,b)
  ret = 'a is greater';
  if (a > b)
    return;
  end
  ret = 'b is greater';
  printf('finishing up...\n');
```

Next we exercise the function with a few simple test cases:

```
--> return_func(1,3)
finishing up...
ans =
  <string>  - size: [1 12]
 b is greater
--> return_func(5,2)
ans =
  <string>  - size: [1 12]
 a is greater
```

In the second example, we take the function and rewrite it to use a `keyboard` statement inside the `if` statement.

```
     return_func2.m
function ret = return_func2(a,b)
  if (a > b)
     ret = 'a is greater';
     keyboard;
  else
     ret = 'b is greater';
  end
  printf('finishing up...\n');
```

Now, we call the function with a larger first argument, which triggers the `keyboard` session. After verifying a few values inside the `keyboard` session, we issue a `return` statement to resume execution.

```
--> return_func2(2,4)
finishing up...
ans =
  <string>  - size: [1 12]
 b is greater
--> return_func2(5,1)
[return_func2,4] --> ret
ans =
  <string>  - size: [1 12]
 a is greater
[return_func2,4] --> a
ans =
  <int32>  - size: [1 1]
 5
[return_func2,4] --> b
ans =
  <int32>  - size: [1 1]
 1
[return_func2,4] --> return
finishing up...
ans =
  <string>  - size: [1 12]
 a is greater
```

5.10 SWITCH Switch statement

5.10.1 Usage

The `switch` statement is used to selective execute code based on the value of either scalar value or a string. The general syntax for a `switch` statement is

```
  switch(expression)
    case test_expression_1
      statements
    case test_expression_2
      statements
    otherwise:
      statements
  end
```

The `otherwise` clause is optional. Note that each test expression can either be a scalar value, a string to test against (if the switch expression is a string), or a `cell-array` of expressions to test against. Note that unlike `C switch` statements, the FreeMat `switch` does not have fall-through, meaning that the statements associated with the first matching case are executed, and then the `switch` ends. Also, if the `switch` expression matches multiple `case` expressions, only the first one is executed.

5.10.2 Examples

Here is an example of a `switch` expression that tests against a string input:

```
     switch_test.m
function c = switch_test(a)
  switch(a)
    case {'lima beans','root beer'}
      c = 'food';
    case {'red','green','blue'}
      c = 'color';
    otherwise
      c = 'not sure';
  end
```

Now we exercise the switch statements

```
--> switch_test('root beer')
ans =
  <string>  - size: [1 4]
 food
--> switch_test('red')
ans =
  <string>  - size: [1 5]
 color
--> switch_test('carpet')
ans =
  <string>  - size: [1 8]
 not sure
```

5.11 TRY-CATCH Try and Catch Statement

5.11.1 Usage

The try and catch statements are used for error handling and control. A concept present in C++, the try and catch statements are used with two statement blocks as follows

```
try
  statements_1
catch
  statements_2
end
```

The meaning of this construction is: try to execute statements_1, and if any errors occur during the execution, then execute the code in statements_2. An error can either be a FreeMat generated error (such as a syntax error in the use of a built in function), or an error raised with the error command.

5.11.2 Examples

Here is an example of a function that uses error control via try and catch to check for failures in fopen.

```
     read_file.m
function c = read_file(filename)
try
   fp = fopen(filename,'r');
   c = fgetline(fp);
   fclose(fp);
catch
   c = ['could not open file because of error :' lasterr]
end
```

Now we try it on an example file - first one that does not exist, and then on one that we create (so that we know it exists).

```
--> read_file('this_filename_is_invalid')
c =
  <string>  - size: [1 63]
 could not open file because of error :No such file or directory
ans =
  <string>  - size: [1 63]
 could not open file because of error :No such file or directory
--> fp = fopen('test_text.txt','w');
--> fprintf(fp,'a line of text\n');
--> fclose(fp);
--> read_file('test_text.txt')
ans =
```

```
  <string>  - size: [1 15]
 a line of text
```

5.12 WARNING Emits a Warning Message

5.12.1 Usage

The `warning` function causes a warning message to be sent to the user. The general syntax for its use is

```
   warning(s)
```

where `s` is the string message containing the warning.

5.13 WHILE While Loop

5.13.1 Usage

The `while` loop executes a set of statements as long as a the test condition remains `true`. The syntax of a `while` loop is

```
  while test_expression
     statements
  end
```

Note that a conditional expression is considered true if the real part of the result of the expression contains any non-zero elements (this strange convention is adopted for compatibility with MATLAB).

5.13.2 Examples

Here is a `while` loop that adds the integers from `1` to `100`:

```
--> accum = 0;
--> k=1;
--> while (k<100), accum = accum + k; k = k + 1; end
--> accum
ans =
  <int32>  - size: [1 1]
 4950
```

Chapter 6

FreeMat Functions

6.1 BUILTIN Evaulate Builtin Function

6.1.1 Usage

The builtin function evaluates a built in function with the given name, bypassing any overloaded functions. The syntax of builtin is

```
[y1,y2,...,yn] = builtin(fname,x1,x2,...,xm)
```

where fname is the name of the function to call. Apart from the fact that fname must be a string, and that builtin always calls the non-overloaded method, it operates exactly like feval. Note that unlike MATLAB, builtin does not force evaluation to an actual compiled function. It simply subverts the activation of overloaded method calls.

6.2 CLOCK Get Current Time

6.2.1 Usage

Returns the current date and time as a vector. The syntax for its use is

```
y = clock
```

where y has the following format:

```
y = [year month day hour minute seconds]
```

6.2.2 Example

Here is the time that this manual was last built:

```
--> clock
ans =
  <double>  - size: [1 6]
```

```
Columns 1 to 3
 2006.0000000000000     6.0000000000000     6.0000000000000

Columns 4 to 6
   21.0000000000000    39.0000000000000    26.3024320602417
```

6.3 CLOCKTOTIME Convert Clock Vector to Epoch Time

6.3.1 Usage

Given the output of the `clock` command, this function computes the epoch time, i.e, the time in seconds since January 1,1970 at 00:00:00 UTC. This function is most useful for calculating elapsed times using the clock, and should be accurate to less than a millisecond (although the true accuracy depends on accuracy of the argument vector). The usage for `clocktotime` is

```
y = clocktotime(x)
```

where `x` must be in the form of the output of `clock`, that is

```
x = [year month day hour minute seconds]
```

6.3.2 Example

Here is an example of using `clocktotime` to time a delay of 1 second

```
--> x = clock
x =
  <double>  - size: [1 6]

Columns 1 to 3
 2006.000000000000000     6.000000000000000     6.000000000000000

Columns 4 to 6
   21.000000000000000    39.000000000000000    26.329164028167725
--> sleep(1)
--> y = clock
y =
  <double>  - size: [1 6]

Columns 1 to 3
 2006.000000000000000     6.000000000000000     6.000000000000000

Columns 4 to 6
   21.000000000000000    39.000000000000000    27.330610036849976
--> clocktotime(y) - clocktotime(x)
ans =
```

```
  <double>  - size: [1 1]
 1.001446008682251
```

6.4 COMPUTER Computer System FreeMat is Running On

6.4.1 Usage

Returns a string describing the name of the system FreeMat is running on. The exact value of this string is subject to change, although the 'MAC' and 'PCWIN' values are probably fixed.

```
str = computer
```

Currently, the following return values are defined

- 'PCWIN' - MS Windows
- 'MAC' - Mac OS X
- 'UNIX' - All others

6.5 EDITOR Open Editor Window

6.5.1 Usage

Brings up the editor window. The `editor` function takes no arguments:

```
editor
```

6.6 ETIME Elapsed Time Function

6.6.1 Usage

The `etime` calculates the elapsed time between two `clock` vectors `x1` and `x2`. The syntax for its use is

```
y = etime(x1,x2)
```

where `x1` and `x2` are in the `clock` output format

```
x = [year month day hour minute seconds]
```

6.6.2 Example

Here we use `etime` as a substitute for `tic` and `toc`

```
--> x1 = clock;
--> sleep(1);
--> x2 = clock;
--> etime(x2,x1);
```

6.7 EVAL Evaluate a String

6.7.1 Usage

The `eval` function evaluates a string. The general syntax for its use is

```
eval(s)
```

where `s` is the string to evaluate. If `s` is an expression (instead of a set of statements), you can assign the output of the `eval` call to one or more variables, via

```
x = eval(s)
[x,y,z] = eval(s)
```

Another form of `eval` allows you to specify an expression or set of statements to execute if an error occurs. In this form, the syntax for `eval` is

```
eval(try_clause,catch_clause),
```

or with return values,

```
x = eval(try_clause,catch_clause)
[x,y,z] = eval(try_clause,catch_clause)
```

These later forms are useful for specifying defaults. Note that both the `try_clause` and `catch_clause` must be expressions, as the equivalent code is

```
try
  [x,y,z] = try_clause
catch
  [x,y,z] = catch_clause
end
```

so that the assignment must make sense in both cases.

6.7.2 Example

Here are some examples of `eval` being used.

```
--> eval('a = 32')
a =
  <int32>  - size: [1 1]
 32
--> b = eval('a')
b =
  <int32>  - size: [1 1]
 32
```

The primary use of the `eval` statement is to enable construction of expressions at run time.

```
--> s = ['b = a' ' + 2']
s =
  <string>  - size: [1 9]
 b = a + 2
--> eval(s)
b =
  <int32>  - size: [1 1]
 34
```

Here we demonstrate the use of the catch-clause to provide a default value

```
--> a = 32
a =
  <int32>  - size: [1 1]
 32
--> b = eval('a','1')
b =
  <int32>  - size: [1 1]
 32
--> b = eval('z','a+1')
b =
  <int32>  - size: [1 1]
 33
```

Note that in the second case, `b` takes the value of 33, indicating that the evaluation of the first expression failed (because `z` is not defined).

6.8 EVALIN Evaluate a String in Workspace

6.8.1 Usage

The `evalin` function is similar to the `eval` function, with an additional argument up front that indicates the workspace that the expressions are to be evaluated in. The various syntaxes for `evalin` are:

```
evalin(workspace,expression)
x = evalin(workspace,expression)
[x,y,z] = evalin(workspace,expression)
evalin(workspace,try_clause,catch_clause)
x = evalin(workspace,try_clause,catch_clause)
[x,y,z] = evalin(workspace,try_clause,catch_clause)
```

The argument `workspace` must be either 'caller' or 'base'. If it is 'caller', then the expression is evaluated in the caller's work space. That does not mean the caller of `evalin`, but the caller of the current function or script. On the other hand if the argument is 'base', then the expression is evaluated in the base work space. See `eval` for details on the use of each variation.

6.9 EXIT Exit Program

6.9.1 Usage

The usage is

```
exit
```

Quits FreeMat. This script is a simple synonym for `quit`.

6.10 FEVAL Evaluate a Function

6.10.1 Usage

The `feval` function executes a function using its name. The syntax of `feval` is

```
[y1,y2,...,yn] = feval(f,x1,x2,...,xm)
```

where `f` is the name of the function to evaluate, and `xi` are the arguments to the function, and `yi` are the return values.

Alternately, `f` can be a function handle to a function (see the section on `function handles` for more information).

6.10.2 Example

Here is an example of using `feval` to call the `cos` function indirectly.

```
--> feval('cos',pi/4)
ans =
  <double>  - size: [1 1]
 0.7071067811865476
```

Now, we call it through a function handle

```
--> c = @cos
c =
  <function ptr array>  - size: [1 1]
 @cos
--> feval(c,pi/4)
ans =
  <double>  - size: [1 1]
 0.7071067811865476
```

6.11 FILESEP Directory Separation Character

6.11.1 Usage

The `filesep` routine returns the character used to separate directory names on the current platform (basically, a forward slash for Windows, and a backward slash for all other OSes). The syntax is simple:

```
x = filesep
```

6.12 HELP Help

6.12.1 Usage

Displays help on a function available in FreeMat. The help function takes one argument:

```
help topic
```

where `topic` is the topic to look for help on. For scripts, the result of running `help` is the contents of the comments at the top of the file. If FreeMat finds no comments, then it simply displays the function declaration.

6.13 HELPWIN Online Help Window

6.13.1 Usage

Brings up the online help window with the FreeMat manual. The `helpwin` function takes no arguments:

```
helpwin
```

6.14 IMPORT Foreign Function Import

6.14.1 Usage

The import function allows you to call functions that are compiled into shared libraries, as if they were FreeMat functions. The usage is

```
import(libraryname,symbol,function,return,arguments)
```

The argument `libraryname` is the name of the library (as a string) to import the function from. The second argument `symbol` (also a string), is the name of the symbol to import from the library. The third argument `function` is the the name of the function after its been imported into Freemat. The fourth argument is a string that specifies the return type of the function. It can take on one of the following types:

- 'uint8' for an unsigned, 8-bit integer.
- 'int8' for a signed, 8-bit integer.
- 'uint16' an unsigned, 16-bit integer.
- 'int16' a signed, 16-bit integer.
- 'uint32' for an unsigned, 32-bit integer.
- 'int32' for a signed, 32-bit integer.

- 'single' for a 32-bit floating point.
- 'double' for a 64-bit floating point.
- 'void' for no return type.

The fourth argument is more complicated. It encodes the arguments of the imported function using a special syntax. In general, the argument list is a string consisting of entries of the form:

```
type[optional bounds check] {optional &}name
```

Here is a list of various scenarios (expressed in 'C'), and the corresponding entries, along with snippets of code.

Scalar variable passed by value: Suppose a function is defined in the library as

```
int fooFunction(float t),
```

i.e., it takes a scalar value (or a string) that is passed by value. Then the corresponding argument string would be

```
'float t'
```

For a C-string, which corresponds to a function prototype of

```
int fooFunction(const char *t),
```

the corresponding argument string would be

```
'string t'
```

Other types are as listed above. Note that FreeMat will automatically promote the type of scalar variables to the type expected by the `C` function. For example, if we call a function expecting a `float` with a `double` or `int16` argument, then FreeMat will automatically apply type promotion rules prior to calling the function.

Scalar variable passed by reference: Suppose a function is defined in the library as

```
int fooFunction(float *t),
```

i.e., it takes a scalar value (or a string) that is passed as a pointer. Then the corresponding argument string would be

```
'float &t'
```

If the function `fooFunction` modifies `t`, then the argument passed in FreeMat will also be modified.

Array variable passed by value: In `C`, it is impossible to distinguish an array being passed from a simple pointer being passed. More often than not, another argument indicates the length of the array. FreeMat has the ability to perform bounds-checking on array values. For example, suppose we have a function of the form

```
int sum_onehundred_ints(int *t),
```

where `sum_onehundred_ints` assumes that `t` is a length `100` vector. Then the corresponding FreeMat argument is

```
'float32[100] t'.
```

Note that because the argument is marked as not being passed by reference, that if `sub_onehundred_ints` modifies the array `t`, this will not affect the FreeMat argument. Note that the bounds-check expression can be any legal scalar expression that evaluates to an integer, and can be a function of the arguments. For example to pass a square $N \times N$ matrix to the following function:

```
float determinantmatrix(int N, float *A),
```

we can use the following argument to `import`:

```
'int32 N, float[N*N] t'.
```

Array variable passed by reference: If the function in `C` modifies an array, and we wish this to be reflected in the FreeMat side, we must pass that argument by reference. Hence, consider the following hypothetical function that squares the elements of an array (functionally equivalent to $x.^2$):

```
void squarearray(int N, float *A)
```

we can use the following argument to `import`:

```
'int32 N, float[N] &A'.
```

Note that to avoid problems with memory allocation, external functions are not allowed to return pointers. As a result, as a general operating mechanism, the FreeMat code must allocate the proper arrays, and then pass them by reference to the external function.

6.14.2 Example

Here is a complete example. We have a `C` function that adds two float vectors of the same length, and stores the result in a third array that is modified by the function. First, the `C` code:

```
     addArrays.c
void addArrays(int N, float *a, float *b, float *c) {
  int i;

  for (i=0;i<N;i++)
   c[i] = a[i] + b[i];
}
```

We then compile this into a dynamic library, say, `add.so`. The import command would then be:

```
import('add.so','addArrays','addArrays','void', ...
       'int32 N, float[N] a, float[N] b, float[N] &c');
```

We could then exercise the function exactly as if it had been written in FreeMat. The following only works on systems using the GNU C Compiler:

```
--> if (strcmp(computer,'MAC')) system('gcc -bundle -flat_namespace -undefined suppress -o
--> if (~strcmp(computer,'MAC')) system('gcc -shared -fPIC -o add.so addArrays.c'); end;
--> import('add.so','addArrays','addArrays','void','int32 N, float[N] a, float[N] b, float
--> a = [3,2,3,1];
--> b = [5,6,0,2];
--> c = [0,0,0,0];
--> addArrays(length(a),a,b,c)
ans =
  <double>  - size: [0 0]
  []
--> c
ans =
  <float>  - size: [1 4]

Columns 1 to 4
 8  8  3  3
```

6.15 LOADLIB Load Library Function

6.15.1 Usage

The `loadlib` function allows a function in an external library to be added to FreeMat dynamically. This interface is generally to be used as last resort, as the form of the function being called is assumed to match the internal implementation. In short, this is not the interface mechanism of choice. For all but very complicated functions, the `import` function is the preferred approach. Thus, only a very brief summary of it is presented here. The syntax for `loadlib` is

```
  loadlib(libfile, symbolname, functionname, nargin, nargout)
```

where `libfile` is the complete path to the library to use, `symbolname` is the name of the symbol in the library, `functionname` is the name of the function after it is imported into FreeMat (this is optional, it defaults to the `symbolname`), `nargin` is the number of input arguments (defaults to 0), and `nargout` is the number of output arguments (defaults to 0). If the number of (input or output) arguments is variable then set the corresponding argument to `-1`.

6.16 MFILENAME Name of Current Function

6.16.1 Usage

Returns a string describing the name of the current function. For M-files this string will be the complete filename of the function. This is true even for subfunctions. The syntax for its use is

```
    y = mfilename
```

6.17 PATH Get or Set FreeMat Path

6.17.1 Usage

The `path` routine has one of the following syntaxes. In the first form

```
x = path
```

`path` simply returns the current path. In the second, the current path is replaced by the argument string `'thepath'`

```
path('thepath')
```

In the third form, a new path is appended to the current search path

```
path(path,'newpath')
```

In the fourth form, a new path is prepended to the current search path

```
path('newpath',path)
```

6.18 PATHSEP Path Directories Separation Character

6.18.1 Usage

The `pathsep` routine returns the character used to separate multiple directories on a path string for the current platform (basically, a semicolon for Windows, and a regular colon for all other OSes). The syntax is simple:

```
x = pathsep
```

6.19 PATHTOOL Open Path Setting Tool

6.19.1 Usage

Brings up the pathtool dialog. The `pathtool` function takes no arguments:

```
pathtool
```

6.20 PCODE Convert a Script or Function to P-Code

6.20.1 Usage

Writes out a script or function as a P-code function. The general syntax for its use is:

```
pcode fun1 fun2 ...
```

The compiled functions are written to the current directory.

6.21 QUIT Quit Program

6.21.1 Usage

The `quit` statement is used to immediately exit the FreeMat application. The syntax for its use is

```
quit
```

6.22 RESCAN Rescan M Files for Changes

6.22.1 Usage

Usually, FreeMat will automatically determine when M Files have changed, and pick up changes you have made to M files. Sometimes, you have to force a refresh. Use the `rescan` command for this purpose. The syntax for its use is

```
rescan
```

6.23 SLEEP Sleep For Specified Number of Seconds

6.23.1 Usage

Suspends execution of FreeMat for the specified number of seconds. The general syntax for its use is

```
sleep(n),
```

where `n` is the number of seconds to wait.

6.24 SOURCE Execute an Arbitrary File

6.24.1 Usage

The `source` function executes the contents of the given filename one line at a time (as if it had been typed at the `-->` prompt). The `source` function syntax is

```
source(filename)
```

where `filename` is a `string` containing the name of the file to process.

6.24.2 Example

First, we write some commands to a file (note that it does not end in the usual `.m` extension):

```
--> fp = fopen('source_test','w');
--> fprintf(fp,'a = 32;\n');
--> fprintf(fp,'b = a;\n');
--> fclose(fp);
```

Now we source the resulting file.

```
--> clear all
--> source source_test
--> who
  Variable Name      Type   Flags             Size
              a     int32                    [1 1]
              b     int32                    [1 1]
```

6.25 TIC Start Stopwatch Timer

6.25.1 Usage

Starts the stopwatch timer, which can be used to time tasks in FreeMat. The `tic` takes no arguments, and returns no outputs. You must use `toc` to get the elapsed time. The usage is

```
  tic
```

6.25.2 Example

Here is an example of timing the solution of a large matrix equation.

```
--> A = rand(100);
--> b = rand(100,1);
--> tic; c = A\b; toc
ans =
  <double>  - size: [1 1]
 0.009175
```

6.26 TOC Stop Stopwatch Timer

6.26.1 Usage

Stop the stopwatch timer, which can be used to time tasks in FreeMat. The `toc` function takes no arguments, and returns no outputs. You must use `toc` to get the elapsed time. The usage is

```
  toc
```

6.26.2 Example

Here is an example of timing the solution of a large matrix equation.

```
--> A = rand(100);
--> b = rand(100,1);
--> tic; c = A\b; toc
ans =
  <double>  - size: [1 1]
 0.005841
```

Chapter 7

Debugging FreeMat Code

7.1 DBAUTO Control Dbauto Functionality

7.1.1 Usage

The dbauto functionality in FreeMat allows you to debug your FreeMat programs. When `dbauto` is `on`, then any error that occurs while the program is running causes FreeMat to stop execution at that point and return you to the command line (just as if you had placed a `keyboard` command there). You can then examine variables, modify them, and resume execution using `return`. Alternately, you can exit out of all running routines via a `retall` statement. Note that errors that occur inside of `try/catch` blocks do not (by design) cause auto breakpoints. The `dbauto` function toggles the dbauto state of FreeMat. The syntax for its use is

```
dbauto(state)
```

where `state` is either

```
dbauto('on')
```

to activate dbauto, or

```
dbauto('off')
```

to deactivate dbauto. Alternately, you can use FreeMat's string-syntax equivalence and enter

```
dbauto on
```

or

```
dbauto off
```

to turn dbauto on or off (respectively). Entering `dbauto` with no arguments returns the current state (either 'on' or 'off').

7.2 DBDELETE Delete a Breakpoint

7.2.1 Usage

The `dbdelete` function deletes a breakpoint. The syntax for the `dbdelete` function is

```
dbdelete(num)
```

where `num` is the number of the breakpoint to delete.

7.3 DBLIST List Breakpoints

7.3.1 Usage

List the current set of breakpoints. The syntax for the `dblist` is simply

```
dblist
```

7.4 DBSTEP Step N Statements

7.4.1 Usage

Step `N` statements during debug mode. The synax for this is either

```
dbstep(N)
```

to step `N` statements, or

```
dbstep
```

to step one statement.

7.5 DBSTOP

7.5.1 Usage

Set a breakpoint. The syntax for this is:

```
dbstop(funcname,linenumber)
```

where `funcname` is the name of the function where we want to set the breakpoint, and `linenumber` is the line number.

Chapter 8

Sparse Matrix Support

8.1 EIGS Sparse Matrix Eigendecomposition

8.1.1 Usage

Computes the eigendecomsition of a sparse square matrix. The `eigs` function has several forms. The most general form is

```
[V,D] = eigs(A,k,sigma)
```

where `A` is the matrix to analyze, `k` is the number of eigenvalues to compute and `sigma` determines which eigenvallues to solve for. Valid values for `sigma` are 'lm' - largest magnitude 'sm' - smallest magnitude 'la' - largest algebraic (for real symmetric problems) 'sa' - smallest algebraic (for real symmetric problems) 'be' - both ends (for real symmetric problems) 'lr' - largest real part 'sr' - smallest real part 'li' - largest imaginary part 'si' - smallest imaginary part scalar - find the eigenvalues closest to `sigma`. The returned matrix `V` contains the eigenvectors, and `D` stores the eigenvalues. The related form

```
d = eigs(A,k,sigma)
```

computes only the eigenvalues and not the eigenvectors. If `sigma` is omitted, as in the forms

```
[V,D] = eigs(A,k)
```

and

```
d = eigs(A,k)
```

then `eigs` returns the largest magnitude eigenvalues (and optionally the associated eigenvectors). As an even simpler form, the forms

```
[V,D] = eigs(A)
```

and

```
d = eigs(A)
```

then `eigs` returns the six largest magnitude eigenvalues of `A` and optionally the eigenvectors. The `eigs` function uses ARPACK to compute the eigenvectors and/or eigenvalues. Note that due to a limitation in the interface into ARPACK from FreeMat, the number of eigenvalues that are to be computed must be strictly smaller than the number of columns (or rows) in the matrix.

8.1.2 Example

Here is an example of using `eigs` to calculate eigenvalues of a matrix, and a comparison of the results with `eig`

```
--> a = sparse(rand(9))
a =
  <double>  - size: [9 9]
Matrix is sparse with 81 nonzeros
--> eigs(a)
ans =
  <dcomplex>  - size: [6 1]

Columns 1 to 1
  4.56344441016545499 0.00000000000000000i
  0.84045857983263339 0.00000000000000000i
 -0.65346421537567945+ 0.16160440340808402i
 -0.65346421537567945-0.16160440340808402i
  0.22030330063491105-0.39352774125459272i
  0.22030330063491105+ 0.39352774125459272i
--> eig(full(a))
ans =
  <dcomplex>  - size: [9 1]

Columns 1 to 1
  4.56344441016545943 0.00000000000000000i
  0.84045857983263283 0.00000000000000000i
  0.44641118312756950 0.00000000000000000i
  0.22030330063491119+ 0.39352774125459289i
  0.22030330063491119-0.39352774125459289i
 -0.65346421537567956+ 0.16160440340808366i
 -0.65346421537567956-0.16160440340808366i
 -0.30933579500854164+ 0.28331426086451600i
 -0.30933579500854164-0.28331426086451600i
```

Next, we exercise some of the variants of `eigs`:

```
--> eigs(a,4,'sm')
ans =
  <dcomplex>  - size: [4 1]

Columns 1 to 1
```

```
 -0.30933579500854180+ 0.28331426086451622i
 -0.30933579500854180-0.28331426086451622i
  0.44641118312756989 0.00000000000000000i
  0.22030330063491127-0.39352774125459294i
--> eigs(a,4,'lr')
ans =
  <dcomplex>  - size: [4 1]

Columns 1 to 1
 4.563444410165453210.00000000000000000i
 0.840458579832632830.00000000000000000i
 0.446411183127570440.00000000000000000i
 0.22030330063491141+0.39352774125459333i
--> eigs(a,4,'sr')
ans =
  <dcomplex>  - size: [4 1]

Columns 1 to 1
 -0.65346421537567900-0.16160440340808357i
 -0.65346421537567900+ 0.16160440340808357i
 -0.30933579500854175-0.28331426086451605i
 -0.30933579500854175+ 0.28331426086451605i
```

8.2 FULL Convert Sparse Matrix to Full Matrix

8.2.1 Usage

Converts a sparse matrix to a full matrix. The syntax for its use is

```
y = full(x)
```

The type of `x` is preserved. Be careful with the function. As a general rule of thumb, if you can work with the `full` representation of a function, you probably do not need the sparse representation.

8.2.2 Example

Here we convert a full matrix to a sparse one, and back again.

```
--> a = [1,0,4,2,0;0,0,0,0,0;0,1,0,0,2]
a =
  <int32>  - size: [3 5]

Columns 1 to 5
 1  0  4  2  0
 0  0  0  0  0
 0  1  0  0  2
--> A = sparse(a)
```

```
A =
  <int32>  - size: [3 5]
Matrix is sparse with 5 nonzeros
--> full(A)
ans =
  <int32>  - size: [3 5]

Columns 1 to 5
 1  0  4  2  0
 0  0  0  0  0
 0  1  0  0  2
```

8.3 NNZ Number of Nonzeros

8.3.1 Usage

Returns the number of nonzero elements in a matrix. The general format for its use is

```
   y = nnz(x)
```

This function returns the number of nonzero elements in a matrix or array. This function works for both sparse and non-sparse arrays. For

8.3.2 Example

```
--> a = [1,0,0,5;0,3,2,0]
a =
  <int32>  - size: [2 4]

Columns 1 to 4
 1  0  0  5
 0  3  2  0
--> nnz(a)
ans =
  <int32>  - size: [1 1]
 4
--> A = sparse(a)
A =
  <int32>  - size: [2 4]
Matrix is sparse with 4 nonzeros
--> nnz(A)
ans =
  <int32>  - size: [1 1]
 4
```

8.4 SPARSE Construct a Sparse Matrix

8.4.1 Usage

Creates a sparse matrix using one of several formats. The first creates a sparse matrix from a full matrix

```
y = sparse(x).
```

The second form creates a sparse matrix containing all zeros that is of the specified size (the sparse equivalent of `zeros`).

```
y = sparse(m,n)
```

where `m` and `n` are integers. Just like the `zeros` function, the sparse matrix returned is of type `float`. The third form constructs a sparse matrix from the IJV syntax. It has two forms. The first version autosizes the sparse matrix

```
y = sparse(i,j,v)
```

while the second version uses an explicit size specification

```
y = sparse(i,j,v,m,n)
```

8.5 SPEYE Sparse Identity Matrix

8.5.1 Usage

Creates a sparse identity matrix of the given size. The syntax for its use is

```
y = speye(m,n)
```

which forms an `m x n` sparse matrix with ones on the main diagonal, or

```
y = speye(n)
```

which forms an `n x n` sparse matrix with ones on the main diagonal. The matrix type is a `float` single precision matrix.

8.5.2 Example

The following creates a 5000 by 5000 identity matrix, which would be difficult to do using `sparse(eye(5000))` because of the large amount of intermediate storage required.

```
--> I = speye(5000)
I =
  <float>  - size: [5000 5000]
Matrix is sparse with 5000 nonzeros
--> full(I(1:10,1:10))
ans =
  <float>  - size: [10 10]
```

```
Columns 1 to 10
 1  0  0  0  0  0  0  0  0  0
 0  1  0  0  0  0  0  0  0  0
 0  0  1  0  0  0  0  0  0  0
 0  0  0  1  0  0  0  0  0  0
 0  0  0  0  1  0  0  0  0  0
 0  0  0  0  0  1  0  0  0  0
 0  0  0  0  0  0  1  0  0  0
 0  0  0  0  0  0  0  1  0  0
 0  0  0  0  0  0  0  0  1  0
 0  0  0  0  0  0  0  0  0  1
```

8.6 SPONES Sparse Ones Function

8.6.1 Usage

Returns a sparse `float` matrix with ones where the argument matrix has nonzero values. The general syntax for it is

```
  y = spones(x)
```

where `x` is a matrix (it may be full or sparse). The output matrix `y` is the same size as `x`, has type `float`, and contains ones in the nonzero positions of `x`.

8.6.2 Examples

Here are some examples of the `spones` function

```
--> a = [1,0,3,0,5;0,0,2,3,0;1,0,0,0,1]
a =
  <int32>  - size: [3 5]

Columns 1 to 5
 1  0  3  0  5
 0  0  2  3  0
 1  0  0  0  1
--> b = spones(a)
b =
  <float>  - size: [3 5]
Matrix is sparse with 7 nonzeros
--> full(b)
ans =
  <float>  - size: [3 5]

Columns 1 to 5
 1  0  1  0  1
```

```
 0  0  1  1  0
 1  0  0  0  1
```

8.7 SPRAND Sparse Uniform Random Matrix

8.7.1 Usage

Creates a sparse matrix with uniformly distributed random entries (on [0,1]). The syntax for its use is

```
  y = sprand(x)
```

where `x` is a sparse matrix, where `y` is a sparse matrix that has random entries where `x` is nonzero. The second form specifies the size of the matrix and the density

```
  y = sprand(m,n,density)
```

where `m` is the number of rows in the output, `n` is the number of columns in the output, and `density` (which is between 0 and 1) is the density of nonzeros in the resulting matrix. Note that for very high densities the actual density of the output matrix may differ from the density you specify. This difference is a result of the way the random entries into the matrix are generated. If you need a very dense random matrix, it is better to generate a full matrix and zero out the entries you do not need.

8.7.2 Examples

Here we seed `sprand` with a full matrix (to demonstrate how the structure of the output is determined by the input matrix when using the first form).

```
--> x = [1,0,0;0,0,1;1,0,0]
x =
  <int32>  - size: [3 3]

Columns 1 to 3
 1  0  0
 0  0  1
 1  0  0
--> y = sprand(x)
y =
  <double>  - size: [3 3]
Matrix is sparse with 3 nonzeros
--> full(y)
ans =
  <double>  - size: [3 3]

Columns 1 to 3
 0.07828198357806448  0.00000000000000000  0.00000000000000000
 0.00000000000000000  0.00000000000000000  0.93876921222238274
 0.21280490445988920  0.00000000000000000  0.00000000000000000
```

The more generic version with a density of `0.001`. On many systems the following is impossible using full matrices

```
--> y = sprand(10000,10000,.001)
y =
  <double>  - size: [10000 10000]
Matrix is sparse with 99946 nonzeros
--> nnz(y)/10000^2
ans =
  <double>  - size: [1 1]
 0.00099946
```

8.8 SPRANDN Sparse Normal Random Matrix

8.8.1 Usage

Creates a sparse matrix with normally distributed random entries (mean 0, sigma 1). The syntax for its use is

```
  y = sprandn(x)
```

where `x` is a sparse matrix, where `y` is a sparse matrix that has random entries where `x` is nonzero. The second form specifies the size of the matrix and the density

```
  y = sprandn(m,n,density)
```

where `m` is the number of rows in the output, `n` is the number of columns in the output, and `density` (which is between 0 and 1) is the density of nonzeros in the resulting matrix. Note that for very high densities the actual density of the output matrix may differ from the density you specify. This difference is a result of the way the random entries into the matrix are generated. If you need a very dense random matrix, it is better to generate a full matrix and zero out the entries you do not need.

8.8.2 Examples

Here we seed `sprandn` with a full matrix (to demonstrate how the structure of the output is determined by the input matrix when using the first form).

```
--> x = [1,0,0;0,0,1;1,0,0]
x =
  <int32>  - size: [3 3]

Columns 1 to 3
 1  0  0
 0  0  1
 1  0  0
--> y = sprandn(x)
y =
  <double>  - size: [3 3]
```

```
Matrix is sparse with 3 nonzeros
--> full(y)
ans =
  <double>  - size: [3 3]

Columns 1 to 3
  0.6236921867754339   0.0000000000000000   0.0000000000000000
  0.0000000000000000   0.0000000000000000   0.9424764236823598
 -1.0286691876982199   0.0000000000000000   0.0000000000000000
```

The more generic version with a density of `0.001`. On many systems the following is impossible using full matrices

```
--> y = sprandn(10000,10000,.001)
y =
  <double>  - size: [10000 10000]
Matrix is sparse with 99953 nonzeros
--> nnz(y)/10000^2
ans =
  <double>  - size: [1 1]
 0.00099953
```

8.9 SPY Visualize Sparsity Pattern of a Sparse Matrix

8.9.1 Usage

Plots the sparsity pattern of a sparse matrix. The syntax for its use is

```
spy(x)
```

which uses a default color and symbol. Alternately, you can use

```
spy(x,colspec)
```

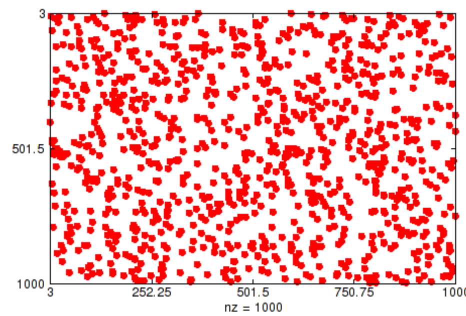
where `colspec` is any valid color and symbol spec accepted by `plot`.

8.9.2 Example

First, an example of a random sparse matrix.

```
--> y = sprand(1000,1000,.001)
y =
  <double>  - size: [1000 1000]
Matrix is sparse with 1000 nonzeros
--> spy(y,'ro')
```

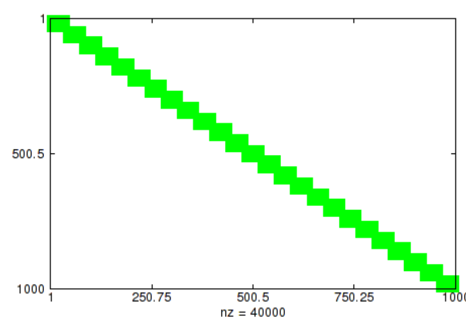
which is shown here



Here is a sparse matrix with a little more structure. First we build a sparse matrix with block diagonal structure, and then use `spy` to visualize the structure.

```
--> A = sparse(1000,1000);
--> for i=1:25; A((1:40) + 40*(i-1),(1:40) + 40*(i-1)) = 1; end;
--> spy(A,'gx')
```

with the result shown here



Chapter 9

Mathematical Functions

9.1 ACOS Inverse Trigonometric Arccosine Function

9.1.1 Usage

Computes the acos function for its argument. The general syntax for its use is

```
y = acos(x)
```

where x is an n-dimensional array of numerical type. Integer types are promoted to the double type prior to calculation of the acos function. Output y is of the same size and type as the input x, (unless x is an integer, in which case y is a double type).

9.1.2 Function Internals

Mathematically, the acos function is defined for all arguments x as

$$\mathrm{acos} x \equiv \frac{pi}{2} + i \log \left(ix + \sqrt{1 - x^2} \right).$$

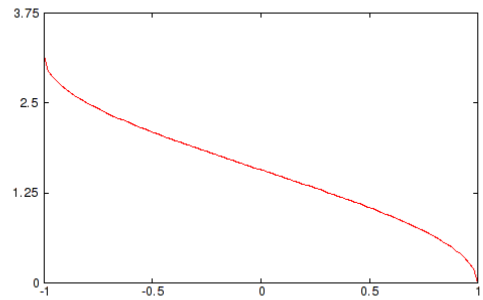
For real valued variables x in the range [-1,1], the function is computed directly using the standard C library's numerical acos function. For both real and complex arguments x, note that generally

$$\mathrm{acos}(\cos(x)) \neq x,$$

9.1.3 Example

The following code demonstates the acos function over the range [-1,1].

```
--> t = linspace(-1,1);
--> plot(t,acos(t))
```



9.2 ANGLE Phase Angle Function

9.2.1 Usage

Compute the phase angle in radians of a complex matrix. The general syntax for its use is

```
p = angle(c)
```

where `c` is an `n`-dimensional array of numerical type.

9.2.2 Function Internals

For a complex number `x`, its polar representation is given by

$$x = |x|e^{j\theta}$$

and we can compute

$$\theta = \mathrm{atan2}(\Im x, \Re x)$$

9.2.3 Example

Here are some examples of the use of `angle` in the polar decomposition of a complex number.

```
--> x = 3+4*i
x =
  <complex>  - size: [1 1]
  3+ 4 i
--> a = abs(x)
a =
  <float>  - size: [1 1]
 5
--> t = angle(x)
t =
```

```
  <float>  - size: [1 1]
 0.9272952
--> a*exp(i*t)
ans =
  <complex>  - size: [1 1]
  3+ 4 i
```

9.3 ASIN Inverse Trigonometric Arcsine Function

9.3.1 Usage

Computes the `asin` function for its argument. The general syntax for its use is

```
  y = asin(x)
```

where `x` is an `n`-dimensional array of numerical type. Integer types are promoted to the `double` type prior to calculation of the `asin` function. Output `y` is of the same size and type as the input `x`, (unless `x` is an integer, in which case `y` is a `double` type).

9.3.2 Function Internals

Mathematically, the `asin` function is defined for all arguments `x` as

$$\mathrm{asin} x \equiv -i \log\left(ix + \sqrt{1-x^2}\right).$$

For real valued variables `x` in the range `[-1,1]`, the function is computed directly using the standard C library's numerical `asin` function. For both real and complex arguments `x`, note that generally

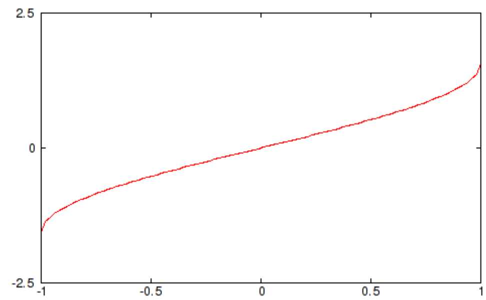
$$\mathrm{asin}(\sin(x)) \neq x,$$

due to the periodicity of `sin(x)`.

9.3.3 Example

The following code demonstates the `asin` function over the range `[-1,1]`.

```
--> t = linspace(-1,1);
--> plot(t,asin(t))
```



9.4 ATAN Inverse Trigonometric Arctangent Function

9.4.1 Usage

Computes the `atan` function for its argument. The general syntax for its use is

```
y = atan(x)
```

where `x` is an `n`-dimensional array of numerical type. Integer types are promoted to the `double` type prior to calculation of the `atan` function. Output `y` is of the same size and type as the input `x`, (unless `x` is an integer, in which case `y` is a `double` type).

9.4.2 Function Internals

Mathematically, the `atan` function is defined for all arguments `x` as

$$\mathrm{atan} x \equiv \frac{i}{2}\left(\log(1-ix) - \log(ix+1)\right).$$

For real valued variables `x`, the function is computed directly using the standard C library's numerical `atan` function. For both real and complex arguments `x`, note that generally

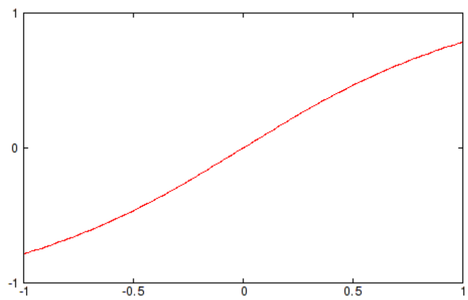
$$\mathrm{atan}(\tan(x)) \neq x,$$

due to the periodicity of `tan(x)`.

9.4.3 Example

The following code demonstates the `atan` function over the range `[-1,1]`.

```
--> t = linspace(-1,1);
--> plot(t,atan(t))
```



9.5 ATAN2 Inverse Trigonometric 4-Quadrant Arctangent Function

9.5.1 Usage

Computes the `atan2` function for its argument. The general syntax for its use is

```
y = atan2(y,x)
```

where `x` and `y` are `n`-dimensional arrays of numerical type. Integer types are promoted to the `double` type prior to calculation of the `atan2` function. The size of the output depends on the size of `x` and `y`. If `x` is a scalar, then `z` is the same size as `y`, and if `y` is a scalar, then `z` is the same size as `x`. The type of the output is equal to the type of —y/x—.

9.5.2 Function Internals

The function is defined (for real values) to return an angle between `-pi` and `pi`. The signs of `x` and `y` are used to find the correct quadrant for the solution. For complex arguments, the two-argument arctangent is computed via

$$\mathrm{atan2}(y,x) \equiv -i \log\left(\frac{x+iy}{\sqrt{x^2+y^2}}\right)$$

For real valued arguments `x,y`, the function is computed directly using the standard C library's numerical `atan2` function. For both real and complex arguments `x`, note that generally

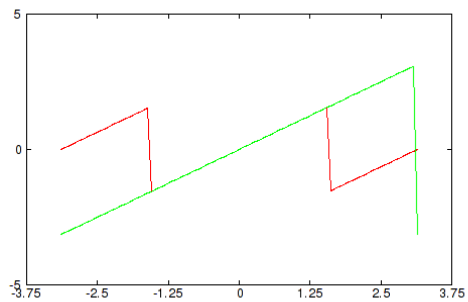
$$\mathrm{atan2}(\sin(x), \cos(x)) \neq x,$$

due to the periodicities of `cos(x)` and `sin(x)`.

9.5.3 Example

The following code demonstates the difference between the `atan2` function and the `atan` function over the range `[-pi,pi]`.

```
--> x = linspace(-pi,pi);
--> sx = sin(x); cx = cos(x);
--> plot(x,atan(sx./cx),x,atan2(sx,cx))
```



Note how the two-argument `atan2` function (green line) correctly "unwraps" the phase of the angle, while the `atan` function (red line) wraps the angle to the interval `[-\pi/2,\pi/2]`.

9.6 COS Trigonometric Cosine Function

9.6.1 Usage

Computes the `cos` function for its argument. The general syntax for its use is

```
y = cos(x)
```

where `x` is an `n`-dimensional array of numerical type. Integer types are promoted to the `double` type prior to calculation of the `cos` function. Output `y` is of the same size and type as the input `x`, (unless `x` is an integer, in which case `y` is a `double` type).

9.6.2 Function Internals

Mathematically, the `cos` function is defined for all real valued arguments `x` by the infinite summation

$$\cos x \equiv \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}.$$

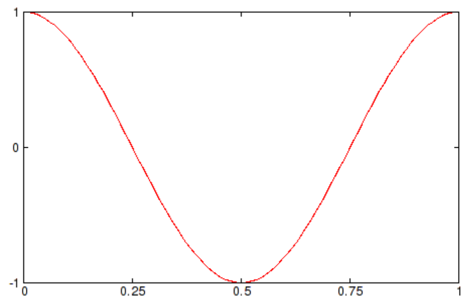
For complex valued arguments `z`, the cosine is computed via

$$\cos z \equiv \cos \Re z \cosh \Im z - \sin \Re z \sinh \Im z.$$

9.6.3 Example

The following piece of code plots the real-valued `cos(2 pi x)` function over one period of `[0,1]`:

```
--> x = linspace(0,1);
--> plot(x,cos(2*pi*x))
```



9.7 COT Trigonometric Cotangent Function

9.7.1 Usage

Computes the `cot` function for its argument. The general syntax for its use is

```
y = cot(x)
```

where `x` is an `n`-dimensional array of numerical type. Integer types are promoted to the `double` type prior to calculation of the `cot` function. Output `y` is of the same size and type as the input `x`, (unless `x` is an integer, in which case `y` is a `double` type).

9.7.2 Function Internals

Mathematically, the `cot` function is defined for all arguments `x` as

$$\cot x \equiv \frac{\cos x}{\sin x}$$

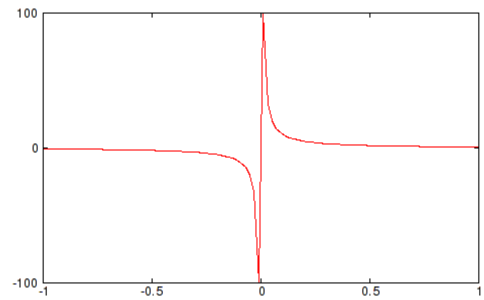
For complex valued arguments `z`, the cotangent is computed via

$$\cot z \equiv \frac{\cos 2\Re z + \cosh 2\Im z}{\sin 2\Re z + i \sinh 2\Im z}.$$

9.7.3 Example

The following piece of code plots the real-valued `cot(x)` function over the interval `[-1,1]`:

```
--> t = linspace(-1,1);
--> plot(t,cot(t))
```



9.8 CROSS Cross Product of Two Vectors

9.8.1 Usage

Computes the cross product of two vectors. The general syntax for its use is

```
c = cross(a,b)
```

where `a` and `b` are 3-element vectors.

9.9 CSC Trigonometric Cosecant Function

9.9.1 Usage

Computes the `csc` function for its argument. The general syntax for its use is

```
y = csc(x)
```

where `x` is an `n`-dimensional array of numerical type. Integer types are promoted to the `double` type prior to calculation of the `csc` function. Output `y` is of the same size and type as the input `x`, (unless `x` is an integer, in which case `y` is a `double` type).

9.9.2 Function Internals

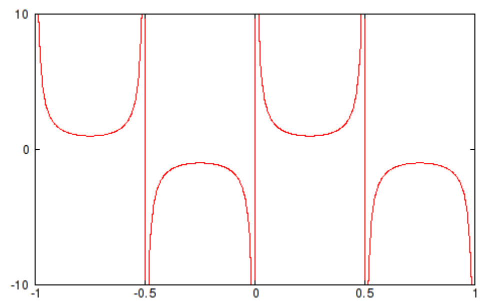
Mathematically, the `csc` function is defined for all arguments as

$$\csc x \equiv \frac{1}{\sin x}.$$

9.9.3 Example

The following piece of code plots the real-valued `csc(2 pi x)` function over the interval of `[-1,1]`:

```
--> t = linspace(-1,1,1000);
--> plot(t,csc(2*pi*t))
--> axis([-1,1,-10,10]);
```



9.10 DAWSON Dawson Integral Function

9.10.1 Usage

Computes the dawson function for real arguments. The `dawson` function takes only a single argument

```
y = dawson(x)
```

where `x` is either a `float` or `double` array. The output vector `y` is the same size (and type) as `x`.

9.10.2 Function Internals

The dawson function is defined as

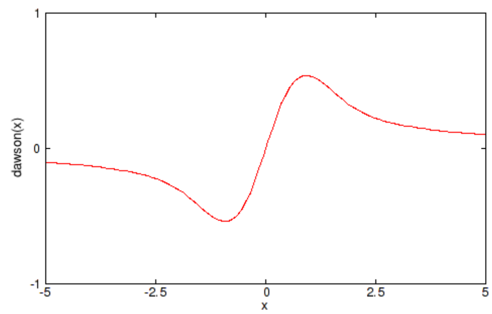
$$\mathrm{dawson}(x) = e^{-x^2} \int_0^x e^{t^2}\, dt$$

9.10.3 Example

Here is a plot of the dawson function over the range `[-5,5]`.

```
--> x = linspace(-5,5);
--> y = dawson(x);
--> plot(x,y); xlabel('x'); ylabel('dawson(x)');
```

which results in the following plot.



9.11 DEG2RAD Convert From Degrees To Radians

9.11.1 Usage

Converts the argument from degrees to radians. The syntax for its use is

```
y = deg2rad(x)
```

where `x` is a numeric array. Conversion is done by simply multiplying `x` by `pi/180`.

9.11.2 Example

How many radians in a circle:

```
--> deg2rad(360) - 2*pi
ans =
  <double>  - size: [1 1]
 0
```

9.12 EI Exponential Integral Function

9.12.1 Usage

Computes the exponential integral function for real arguments. The `ei` function takes only a single argument

```
y = ei(x)
```

where `x` is either a `float` or `double` array. The output vector `y` is the same size (and type) as `x`.

9.12.2 Function Internals

The ei function is defined by the integral:

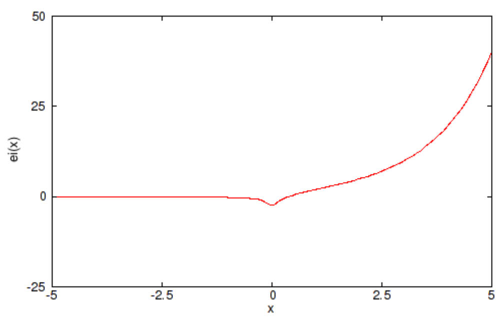
$$\mathrm{ei}(x) = -\int_{-x}^{\infty} \frac{e^{-t}\,dt}{t}.$$

9.12.3 Example

Here is a plot of the `ei` function over the range `[-5,5]`.

```
--> x = linspace(-5,5);
--> y = ei(x);
--> plot(x,y); xlabel('x'); ylabel('ei(x)');
```

which results in the following plot.



9.13 EONE Exponential Integral Function

9.13.1 Usage

Computes the exponential integral function for real arguments. The `eone` function takes only a single argument

```
y = eone(x)
```

where `x` is either a `float` or `double` array. The output vector `y` is the same size (and type) as `x`.

9.13.2 Function Internals

The eone function is defined by the integral:

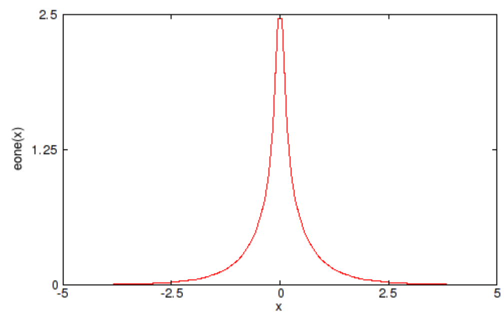
$$\mathrm{eone}(x) = \int_{x}^{\infty} \frac{e^{-u}\,du}{u}.$$

9.13.3 Example

Here is a plot of the `eone` function over the range `[-5,5]`.

```
--> x = linspace(-5,5);
--> y = eone(x);
--> plot(x,y); xlabel('x'); ylabel('eone(x)');
```

which results in the following plot.



9.14 ERF Error Function

9.14.1 Usage

Computes the error function for real arguments. The `erf` function takes only a single argument

```
y = erf(x)
```

where `x` is either a `float` or `double` array. The output vector `y` is the same size (and type) as `x`.

9.14.2 Function Internals

The erf function is defined by the integral:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt,$$

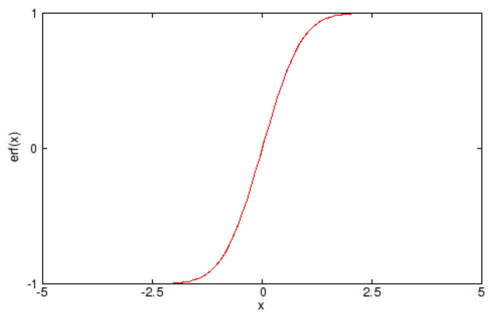
and is the integral of the normal distribution.

9.14.3 Example

Here is a plot of the erf function over the range `[-5,5]`.

```
--> x = linspace(-5,5);
--> y = erf(x);
--> plot(x,y); xlabel('x'); ylabel('erf(x)');
```

which results in the following plot.



9.15 ERFC Complimentary Error Function

9.15.1 Usage

Computes the complimentary error function for real arguments. The `erfc` function takes only a single argument

```
y = erfc(x)
```

where `x` is either a `float` or `double` array. The output vector `y` is the same size (and type) as `x`.

9.15.2 Function Internals

The erfc function is defined by the integral:

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_{x}^{\infty} e^{-t^2}\, dt,$$

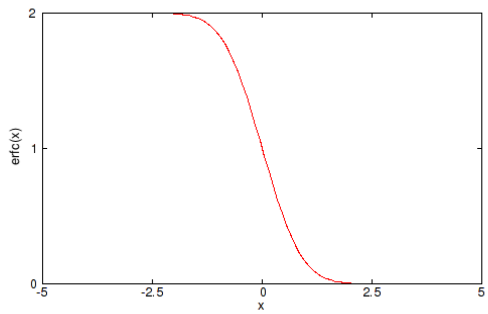
and is the integral of the normal distribution.

9.15.3 Example

Here is a plot of the `erfc` function over the range `[-5,5]`.

```
--> x = linspace(-5,5);
--> y = erfc(x);
--> plot(x,y); xlabel('x'); ylabel('erfc(x)');
```

which results in the following plot.



9.16 ERFCX Complimentary Weighted Error Function

9.16.1 Usage

Computes the complimentary error function for real arguments. The `erfcx` function takes only a single argument

```
y = erfcx(x)
```

where `x` is either a `float` or `double` array. The output vector `y` is the same size (and type) as `x`.

9.16.2 Function Internals

The erfcx function is defined by the integral:

$$\mathrm{erfcx}(x) = \frac{2e^{x^2}}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\, dt,$$

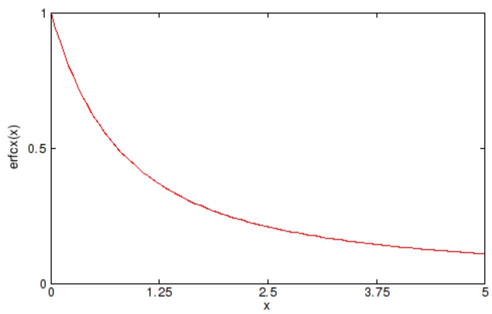
and is an exponentially weighted integral of the normal distribution.

9.16.3 Example

Here is a plot of the `erfcx` function over the range `[-5,5]`.

```
--> x = linspace(0,5);
--> y = erfcx(x);
--> plot(x,y); xlabel('x'); ylabel('erfcx(x)');
```

which results in the following plot.



9.17 EXP Exponential Function

9.17.1 Usage

Computes the `exp` function for its argument. The general syntax for its use is

```
y = exp(x)
```

where `x` is an `n`-dimensional array of numerical type. Integer types are promoted to the `double` type prior to calculation of the `exp` function. Output `y` is of the same size and type as the input `x`, (unless `x` is an integer, in which case `y` is a `double` type).

9.17.2 Function Internals

Mathematically, the `exp` function is defined for all real valued arguments `x` as

$$\exp x \equiv e^x,$$

where

$$e = \sum_0^\infty \frac{1}{k!}$$

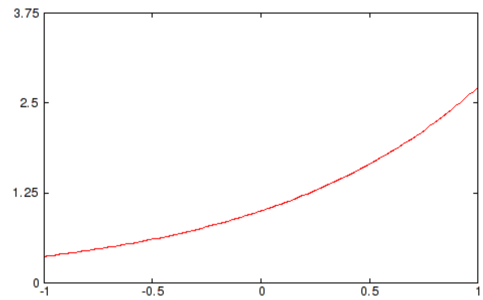
and is approximately `2.718281828459045` (returned by the function `e`). For complex values `z`, the famous Euler formula is used to calculate the exponential

$$e^z = e^{|z|}\left[\cos \Re z + i \sin \Re z\right]$$

9.17.3 Example

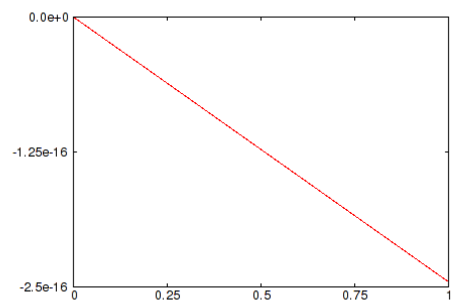
The following piece of code plots the real-valued `exp` function over the interval `[-1,1]`:

```
--> x = linspace(-1,1);
--> plot(x,exp(x))
```



In the second example, we plot the unit circle in the complex plane `e^{i 2 pi x}` for `x in [-1,1]`.

```
--> x = linspace(-1,1);
--> plot(exp(-i*x*2*pi))
```



9.18 EXPEI Exponential Weighted Integral Function

9.18.1 Usage

Computes the exponential weighted integral function for real arguments. The `expei` function takes only a single argument

```
y = expei(x)
```

where `x` is either a `float` or `double` array. The output vector `y` is the same size (and type) as `x`.

9.18.2 Function Internals

The expei function is defined by the integral:

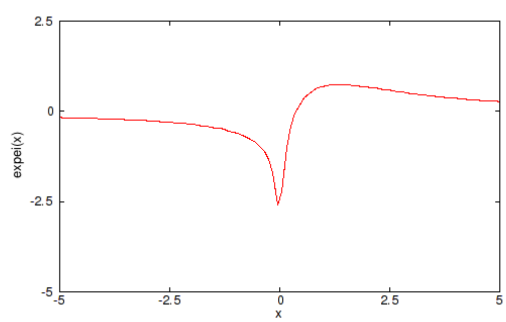
$$\mathrm{expei}(x) = -e^{-x} \int_{-x}^{\infty} \frac{e^{-t}\,dt}{t}.$$

9.18.3 Example

Here is a plot of the `expei` function over the range `[-5,5]`.

```
--> x = linspace(-5,5);
--> y = expei(x);
--> plot(x,y); xlabel('x'); ylabel('expei(x)');
```

which results in the following plot.



9.19 FIX Round Towards Zero

9.19.1 Usage

Rounds the argument array towards zero. The syntax for its use is

```
y = fix(x)
```

where `x` is a numeric array. For positive elements of `x`, the output is the largest integer smaller than `x`. For negative elements of `x` the output is the smallest integer larger than `x`. For complex `x`, the operation is applied seperately to the real and imaginary parts.

9.19.2 Example

Here is a simple example of the `fix` operation on some values

```
--> a = [-1.8,pi,8,-pi,-0.001,2.3+0.3i]
a =
  <dcomplex>  - size: [1 6]

Columns 1 to 1
 -1.800000000000000 0.000000000000000i

Columns 2 to 2
  3.141592653589793 0.000000000000000i

Columns 3 to 3
  8.000000000000000 0.000000000000000i

Columns 4 to 4
 -3.141592653589793 0.000000000000000i

Columns 5 to 5
 -0.001000000000000 0.000000000000000i

Columns 6 to 6
  2.300000000000000+ 0.300000000000000i
--> fix(a)
ans =
  <dcomplex>  - size: [1 6]

Columns 1 to 6
  -1  0i    3  0i    8  0i   -3  0i    0  0i    2  0i
```

9.20 GAMMA Gamma Function

9.20.1 Usage

Computes the gamma function for real arguments. The `gamma` function takes only a single argument

```
  y = gamma(x)
```

where `x` is either a `float` or `double` array. The output vector `y` is the same size (and type) as `x`.

9.20.2 Function Internals

The gamma function is defined by the integral:

$$\Gamma(x) = \int_0^\infty e^{-t} t^{x-1}\, dt$$

The gamma function obeys the interesting relationship

$$\Gamma(x) = (x-1)\Gamma(x-1),$$

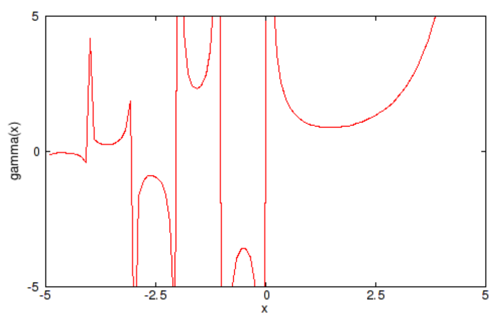
and for integer arguments, is equivalent to the factorial function.

9.20.3 Example

Here is a plot of the gamma function over the range [-5,5].

```
--> x = linspace(-5,5);
--> y = gamma(x);
--> plot(x,y); xlabel('x'); ylabel('gamma(x)');
--> axis([-5,5,-5,5]);
```

which results in the following plot.



9.21 GAMMALN Log Gamma Function

9.21.1 Usage

Computes the natural log of the gamma function for real arguments. The `gammaln` function takes only a single argument

```
y = gammaln(x)
```

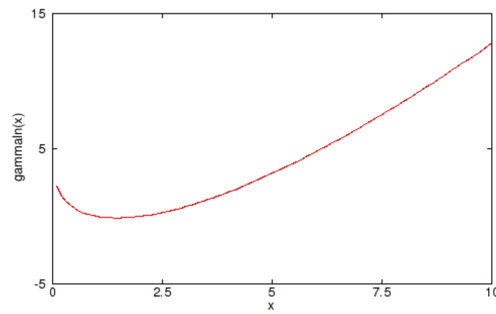
where `x` is either a `float` or `double` array. The output vector `y` is the same size (and type) as `x`.

9.21.2 Example

Here is a plot of the `gammaln` function over the range [-5,5].

```
--> x = linspace(0,10);
--> y = gammaln(x);
--> plot(x,y); xlabel('x'); ylabel('gammaln(x)');
```

which results in the following plot.



9.22 IDIV Integer Division Operation

9.22.1 Usage

Computes the integer division of two arrays. The syntax for its use is

```
   y = idiv(a,b)
```

where `a` and `b` are arrays or scalars. The effect of the `idiv` is to compute the integer division of `b` into `a`.

9.22.2 Example

The following examples show some uses of `idiv` arrays.

```
--> idiv(27,6)
ans =
  <int32>  - size: [1 1]
 4
--> idiv(4,-2)
ans =
  <int32>  - size: [1 1]
 -2
--> idiv(15,3)
ans =
  <int32>  - size: [1 1]
 5
```

9.23 LOG Natural Logarithm Function

9.23.1 Usage

Computes the `log` function for its argument. The general syntax for its use is

```
y = log(x)
```

where `x` is an `n`-dimensional array of numerical type. Integer types are promoted to the `double` type prior to calculation of the `log` function. Output `y` is of the same size as the input `x`. For strictly positive, real inputs, the output type is the same as the input. For negative and complex arguments, the output is complex.

9.23.2 Function Internals

Mathematically, the `log` function is defined for all real valued arguments `x` by the integral

$$\log x \equiv \int_1^x \frac{d\,t}{t}.$$

For complex-valued arguments, `z`, the complex logarithm is defined as

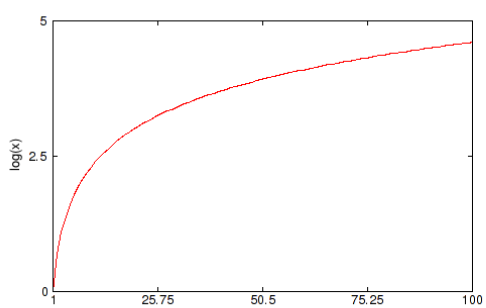
$$\log z \equiv \log |z| + i \arg z,$$

where `arg` is the complex argument of `z`.

9.23.3 Example

The following piece of code plots the real-valued `log` function over the interval `[1,100]`:

```
--> x = linspace(1,100);
--> plot(x,log(x))
--> xlabel('x');
--> ylabel('log(x)');
```



9.24 LOG10 Base-10 Logarithm Function

9.24.1 Usage

Computes the `log10` function for its argument. The general syntax for its use is

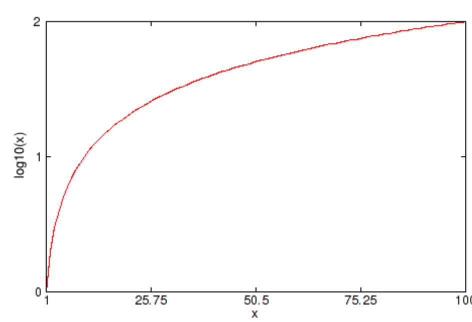
```
y = log10(x)
```

where `x` is an `n`-dimensional array of numerical type. Integer types are promoted to the `double` type prior to calculation of the `log10` function. Output `y` is of the same size as the input `x`. For strictly positive, real inputs, the output type is the same as the input. For negative and complex arguments, the output is complex.

9.24.2 Example

The following piece of code plots the real-valued `log10` function over the interval `[1,100]`:

```
--> x = linspace(1,100);
--> plot(x,log10(x))
--> xlabel('x');
--> ylabel('log10(x)');
```



9.25 LOG2 Base-2 Logarithm Function

9.25.1 Usage

Computes the `log2` function for its argument. The general syntax for its use is

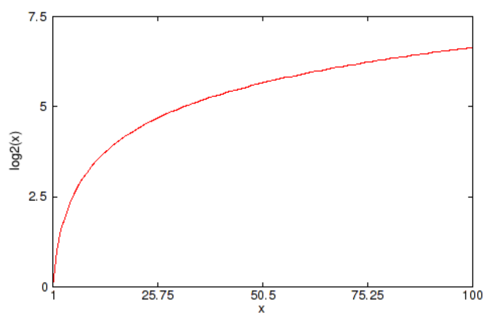
```
y = log2(x)
```

where `x` is an `n`-dimensional array of numerical type. Integer types are promoted to the `double` type prior to calculation of the `log2` function. Output `y` is of the same size as the input `x`. For strictly positive, real inputs, the output type is the same as the input. For negative and complex arguments, the output is complex.

9.25.2 Example

The following piece of code plots the real-valued `log2` function over the interval `[1,100]`:

```
--> x = linspace(1,100);
--> plot(x,log2(x))
--> xlabel('x');
--> ylabel('log2(x)');
```



9.26 MOD Modulus Operation

9.26.1 Usage

Computes the modulus of an array. The syntax for its use is

```
y = mod(x,n)
```

where `x` is matrix, and `n` is the base of the modulus. The effect of the `mod` operator is to add or subtract multiples of `n` to the vector `x` so that each element `x_i` is between `0` and `n` (strictly). Note that `n` does not have to be an integer. Also, `n` can either be a scalar (same base for all elements of `x`), or a vector (different base for each element of `x`).

Note that the following are defined behaviors:

1. `mod(x,0) = x`@
2. `mod(x,x) = 0`@
3. `mod(x,n)`@ has the same sign as `n` for all other cases.

9.26.2 Example

The following examples show some uses of `mod` arrays.

```
--> mod(18,12)
ans =
  <double>  - size: [1 1]
 6
--> mod(6,5)
ans =
  <double>  - size: [1 1]
 1
--> mod(2*pi,pi)
ans =
  <double>  - size: [1 1]
 0
```

Here is an example of using `mod` to determine if integers are even or odd:

```
--> mod([1,3,5,2],2)
ans =
  <double>  - size: [1 4]

Columns 1 to 4
 1  1  1  0
```

Here we use the second form of `mod`, with each element using a separate base.

```
--> mod([9 3 2 0],[1 0 2 2])
ans =
  <double>  - size: [1 4]

Columns 1 to 4
 0  3  0  0
```

9.27 PSI Psi Function

9.27.1 Usage

Computes the psi function for real arguments. The `psi` function takes only a single argument

```
  y = psi(x)
```

where `x` is either a `float` or `double` array. The output vector `y` is the same size (and type) as `x`.

9.27.2 Function Internals

The psi function is defined as

$$\frac{d}{dx}\ln\gamma(x)$$

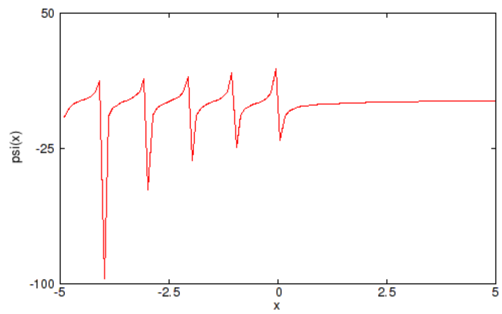
and for integer arguments, is equivalent to the factorial function.

9.27.3 Example

Here is a plot of the psi function over the range [-5,5].

```
--> x = linspace(-5,5);
--> y = psi(x);
--> plot(x,y); xlabel('x'); ylabel('psi(x)');
```

which results in the following plot.



9.28 RAD2DEG Convert From Degrees To Radians

9.28.1 Usage

Converts the argument from radians to degrees. The syntax for its use is

```
y = rad2deg(x)
```

where `x` is a numeric array. Conversion is done by simply multiplying `x` by `180/pi`.

9.28.2 Example

How many degrees in a circle:

```
--> rad2deg(2*pi)
ans =
  <double>  - size: [1 1]
 360
```

9.29 REM Remainder After Division

9.29.1 Usage

Computes the remainder after division of an array. The syntax for its use is

```
y = rem(x,n)
```

where `x` is matrix, and `n` is the base of the modulus. The effect of the `rem` operator is to add or subtract multiples of `n` to the vector `x` so that each element `x_i` is between `0` and `n` (strictly). Note that `n` does not have to be an integer. Also, `n` can either be a scalar (same base for all elements of `x`), or a vector (different base for each element of `x`).

Note that the following are defined behaviors:

1. `rem(x,0) = nan`@
2. `rem(x,x) = 0`@ for nonzero `x`
3. `rem(x,n)`@ has the same sign as `x` for all other cases.

Note that `rem` and `mod` return the same value if `x` and `n` are of the same sign. But differ by `n` if `x` and `y` have different signs.

9.29.2 Example

The following examples show some uses of `rem` arrays.

```
--> rem(18,12)
ans =
  <double>  - size: [1 1]
 6
--> rem(6,5)
ans =
  <double>  - size: [1 1]
 1
--> rem(2*pi,pi)
ans =
  <double>  - size: [1 1]
 0
```

Here is an example of using `rem` to determine if integers are even or odd:

```
--> rem([1,3,5,2],2)
ans =
  <double>  - size: [1 4]

Columns 1 to 4
 1  1  1  0
```

Here we use the second form of `rem`, with each element using a separate base.

```
--> rem([9 3 2 0],[1 0 2 2])
ans =
  <double>  - size: [1 4]

Columns 1 to 4
   0  nan    0    0
```

9.30 SEC Trigonometric Secant Function

9.30.1 Usage

Computes the `sec` function for its argument. The general syntax for its use is

```
y = sec(x)
```

where `x` is an `n`-dimensional array of numerical type. Integer types are promoted to the `double` type prior to calculation of the `sec` function. Output `y` is of the same size and type as the input `x`, (unless `x` is an integer, in which case `y` is a `double` type).

9.30.2 Function Internals

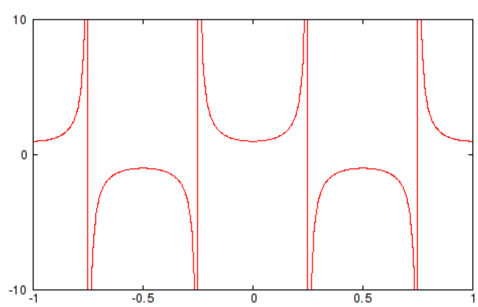
Mathematically, the `sec` function is defined for all arguments as

$$\sec x \equiv \frac{1}{\cos x}.$$

9.30.3 Example

The following piece of code plots the real-valued `sec(2 pi x)` function over the interval of `[-1,1]`:

```
--> t = linspace(-1,1,1000);
--> plot(t,sec(2*pi*t))
--> axis([-1,1,-10,10]);
```



9.31 SIN Trigonometric Sine Function

9.31.1 Usage

Computes the `sin` function for its argument. The general syntax for its use is

```
y = sin(x)
```

where `x` is an `n`-dimensional array of numerical type. Integer types are promoted to the `double` type prior to calculation of the `sin` function. Output `y` is of the same size and type as the input `x`, (unless `x` is an integer, in which case `y` is a `double` type).

9.31.2 Function Internals

Mathematically, the `sin` function is defined for all real valued arguments `x` by the infinite summation

$$\sin x \equiv \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!}.$$

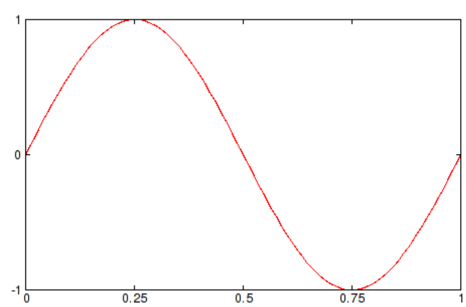
For complex valued arguments `z`, the sine is computed via

$$\sin z \equiv \sin \Re z \cosh \Im z - i \cos \Re z \sinh \Im z.$$

9.31.3 Example

The following piece of code plots the real-valued `sin(2 pi x)` function over one period of `[0,1]`:

```
--> x = linspace(0,1);
--> plot(x,sin(2*pi*x))
```



9.32 SQRT Square Root of an Array

9.32.1 Usage

Computes the square root of the argument matrix. The general syntax for its use is

```
   y = sqrt(x)
```

where `x` is an N-dimensional numerical array.

9.32.2 Example

Here are some examples of using `sqrt`

```
--> sqrt(9)
ans =
  <double>  - size: [1 1]
 3
--> sqrt(i)
ans =
  <complex>  - size: [1 1]
 0.70710677+0.70710677 i
--> sqrt(-1)
ans =
  <dcomplex>  - size: [1 1]
 6.123031769111886e-17+1.000000000000000e+00i
--> x = rand(4)
x =
  <double>  - size: [4 4]

Columns 1 to 3
 0.30321237556395952  0.88194882640444583  0.08364828321231343
 0.89000870990351189  0.28718737336883804  0.85652864497748205
 0.24228626078925408  0.60644730246736522  0.38520056617787590
 0.64087763657404362  0.18691431309757034  0.92958575403063803

Columns 4 to 4
 0.64442638307431366
 0.70020763274943776
 0.58484919732966401
 0.84041505600660127
--> sqrt(x)
ans =
  <double>  - size: [4 4]

Columns 1 to 3
 0.55064723332089804  0.93912130547892791  0.28922012933458391
 0.94340272943399517  0.53589865960724148  0.92548832784507984
 0.49222582296061435  0.77874726482175538  0.62064528208782499
 0.80054833493927380  0.43233587995627931  0.96415027564723443

Columns 4 to 4
 0.80276172247704591
 0.83678410163520545
 0.76475433789529035
 0.91674154264252761
```

9.33 TAN Trigonometric Tangent Function

9.33.1 Usage

Computes the `tan` function for its argument. The general syntax for its use is

```
y = tan(x)
```

where `x` is an `n`-dimensional array of numerical type. Integer types are promoted to the `double` type prior to calculation of the `tan` function. Output `y` is of the same size and type as the input `x`, (unless `x` is an integer, in which case `y` is a `double` type).

9.33.2 Function Internals

Mathematically, the `tan` function is defined for all real valued arguments `x` by the infinite summation

$$\tan x \equiv x + \frac{x^3}{3} + \frac{2x^5}{15} + \cdots,$$

or alternately by the ratio

$$\tan x \equiv \frac{\sin x}{\cos x}$$

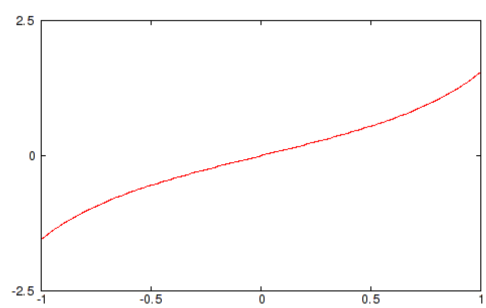
For complex valued arguments `z`, the tangent is computed via

$$\tan z \equiv \frac{\sin 2\Re z + i \sinh 2\Im z}{\cos 2\Re z + \cosh 2\Im z}.$$

9.33.3 Example

The following piece of code plots the real-valued `tan(x)` function over the interval `[-1,1]`:

```
--> t = linspace(-1,1);
--> plot(t,tan(t))
```



Chapter 10

Base Constants

10.1 E Euler Constant (Base of Natural Logarithm)

10.1.1 Usage

Returns a `double` (64-bit floating point number) value that represents Euler's constant, the base of the natural logarithm. Typical usage

```
y = e
```

This value is approximately 2.718281828459045.

10.1.2 Example

The following example demonstrates the use of the `e` function.

```
--> e
ans =
  <double>  - size: [1 1]
 2.718281828459045
--> log(e)
ans =
  <double>  - size: [1 1]
 1
```

10.2 EPS Double Precision Floating Point Relative Machine Precision Epsilon

10.2.1 Usage

Returns `eps`, which quantifies the relative machine precision of floating point numbers (a machine specific quantity). The syntax for `eps` is:

```
y = eps
```

which returns `eps` for `double` precision values. For most typical processors, this value is approximately `2^-52`, or 2.2204e-16.

10.2.2 Example

The following example demonstrates the use of the `eps` function, and one of its numerical consequences.

```
--> eps
ans =
  <double>  - size: [1 1]
 1.1102230246251565e-16
--> 1.0+eps
ans =
  <double>  - size: [1 1]
 1
```

10.3 FEPS Single Precision Floating Point Relative Machine Precision Epsilon

10.3.1 Usage

Returns `feps`, which quantifies the relative machine precision of floating point numbers (a machine specific quantity). The syntax for `feps` is:

```
y = feps
```

which returns `feps` for `single` precision values. For most typical processors, this value is approximately `2^-24`, or 5.9604e-8.

10.3.2 Example

The following example demonstrates the use of the `feps` function, and one of its numerical consequences.

```
--> feps
ans =
  <float>  - size: [1 1]
 0.000000059604645
--> 1.0f+eps
ans =
  <double>  - size: [1 1]
 1
```

10.4 I-J Square Root of Negative One

10.4.1 Usage

Returns a `complex` value that represents the square root of -1. There are two functions that return the same value:

```
y = i
```

and

```
y = j.
```

This allows either `i` or `j` to be used as loop indices. The returned value is a 32-bit complex value.

10.4.2 Example

The following examples demonstrate a few calculations with `i`.

```
--> i
ans =
  <complex>  - size: [1 1]
  0+ 1 i
--> i^2
ans =
  <complex>  - size: [1 1]
  -1  0 i
```

The same calculations with `j`:

```
--> j
ans =
  <complex>  - size: [1 1]
  0+ 1 i
--> j^2
ans =
  <complex>  - size: [1 1]
  -1  0 i
```

Here is an example of how `i` can be used as a loop index and then recovered as the square root of -1.

```
--> accum = 0; for i=1:100; accum = accum + i; end; accum
ans =
  <int32>  - size: [1 1]
 5050
--> i
ans =
  <int32>  - size: [1 1]
 100
```

```
--> clear i
--> i
ans =
  <complex>  - size: [1 1]
  0+ 1 i
```

10.5 INF Infinity Constant

10.5.1 Usage

Returns a value that represents positive infinity for both 32 and 64-bit floating point values.

```
y = inf
```

The returned type is a 32-bit float, but promotion to 64 bits preserves the infinity.

10.5.2 Function Internals

The infinity constant has several interesting properties. In particular:

$$\begin{aligned}
\infty \times 0 &= \mathrm{NaN} \\
\infty \times a &= \infty \, \mathrm{forall} \, a > 0 \\
\infty \times a &= -\infty \, \mathrm{forall} \, a < 0 \\
\infty / \infty &= \mathrm{NaN} \\
\infty / 0 &= \infty
\end{aligned}$$

Note that infinities are not preserved under type conversion to integer types (see the examples below).

10.5.3 Example

The following examples demonstrate the various properties of the infinity constant.

```
--> inf*0
ans =
  <float>  - size: [1 1]
                nan
--> inf*2
ans =
  <float>  - size: [1 1]
 inf
--> inf*-2
ans =
  <float>  - size: [1 1]
 -inf
--> inf/inf
ans =
```

```
  <float>  - size: [1 1]
                       nan
--> inf/0
ans =
  <float>  - size: [1 1]
 inf
--> inf/nan
ans =
  <float>  - size: [1 1]
                       nan
```

Note that infinities are preserved under type conversion to floating point types (i.e., `float`, `double`, `complex` and `dcomplex` types), but not integer types.

```
--> uint32(inf)
ans =
  <uint32>  - size: [1 1]
 0
--> complex(inf)
ans =
  <complex>  - size: [1 1]
  inf   0 i
```

10.6 NAN Not-a-Number Constant

10.6.1 Usage

Returns a value that represents "not-a-number" for both 32 and 64-bit floating point values. This constant is meant to represent the result of arithmetic operations whose output cannot be meaningfully defined (like zero divided by zero).

```
   y = nan
```

The returned type is a 32-bit float, but promotion to 64 bits preserves the not-a-number. The not-a-number constant has one simple property. In particular, any arithmetic operation with a `NaN` results in a `NaN`. These calculations run significantly slower than calculations involving finite quantities! Make sure that you use `NaN`s in extreme circumstances only. Note that `NaN` is not preserved under type conversion to integer types (see the examples below).

10.6.2 Example

The following examples demonstrate a few calculations with the not-a-number constant.

```
--> nan*0
ans =
  <float>  - size: [1 1]
               nan
```

```
--> nan-nan
ans =
  <float>  - size: [1 1]
                 nan
```

Note that NaNs are preserved under type conversion to floating point types (i.e., `float`, `double`, `complex` and `dcomplex` types), but not integer types.

```
--> uint32(nan)
ans =
  <uint32>  - size: [1 1]
 0
--> complex(nan)
ans =
  <complex>  - size: [1 1]
                                   nan                                     0 i
```

10.7 PI Constant Pi

10.7.1 Usage

Returns a `double` (64-bit floating point number) value that represents pi (ratio between the circumference and diameter of a circle...). Typical usage

```
   y = pi
```

This value is approximately 3.141592653589793.

10.7.2 Example

The following example demonstrates the use of the `pi` function.

```
--> pi
ans =
  <double>  - size: [1 1]
 3.141592653589793
--> cos(pi)
ans =
  <double>  - size: [1 1]
 -1
```

10.8 TEPS Type-based Epsilon Calculation

10.8.1 Usage

Returns `eps` for double precision arguments and `feps` for single precision arguments. The syntax for `teps` is

```
y = teps(x)
```

The `teps` function is most useful if you need to compute epsilon based on the type of the array.

10.8.2 Example

The following example demonstrates the use of the `teps` function, and one of its numerical consequences.

```
--> teps(float(3.4))
ans =
  <float>  - size: [1 1]
 0.000000059604645
--> teps(complex(3.4+i*2))
ans =
  <float>  - size: [1 1]
 0.000000059604645
--> teps(double(3.4))
ans =
  <double>  - size: [1 1]
 1.1102230246251565e-16
--> teps(dcomplex(3.4+i*2))
ans =
  <double>  - size: [1 1]
 1.1102230246251565e-16
```

Chapter 11

Elementary Functions

11.1 ABS Absolute Value Function

11.1.1 Usage

Returns the absolute value of the input array for all elements. The general syntax for its use is

```
y = abs(x)
```

where `x` is an `n`-dimensional array of numerical type. The output is the same numerical type as the input, unless the input is `complex` or `dcomplex`. For `complex` inputs, the absolute value is a floating point array, so that the return type is `float`. For `dcomplex` inputs, the absolute value is a double precision floating point array, so that the return type is `double`.

11.1.2 Example

The following demonstrates the `abs` applied to a complex scalar.

```
--> abs(3+4*i)
ans =
  <float>  - size: [1 1]
 5
```

The `abs` function applied to integer and real values:

```
--> abs([-2,3,-4,5])
ans =
  <int32>  - size: [1 4]

Columns 1 to 4
 2  3  4  5
```

For a double-precision complex array,

```
--> abs([2.0+3.0*i,i])
ans =
  <double>  - size: [1 2]

Columns 1 to 2
 3.605551275463989  1.000000000000000
```

11.2 ALL All True Function

11.2.1 Usage

Reduces a logical array along a given dimension by testing for all logical 1s. The general syntax for its use is

```
  y = all(x,d)
```

where `x` is an `n`-dimensions array of `logical` type. The output is of `logical` type. The argument `d` is optional, and denotes the dimension along which to operate. The output `y` is the same size as `x`, except that it is singular along the operated direction. So, for example, if `x` is a `3 x 3 x 4` array, and we `all` operation along dimension `d=2`, then the output is of size `3 x 1 x 4`.

11.2.2 Function Internals

The output is computed via

$$y(m_1,\ldots,m_{d-1},1,m_{d+1},\ldots,m_p) = \min_k x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p)$$

If `d` is omitted, then the minimum is taken over all elements of `x`.

11.2.3 Example

The following piece of code demonstrates various uses of the `all` function

```
--> A = [1,0,0;1,0,0;0,0,1]
A =
  <int32>  - size: [3 3]

Columns 1 to 3
 1  0  0
 1  0  0
 0  0  1
```

We start by calling `all` without a dimension argument, in which case it defaults to testing all values of `A`

```
--> all(A)
ans =
  <logical>  - size: [1 1]
 0
```

The `all` function is useful in expressions also.

```
--> all(A>=0)
ans =
  <logical>  - size: [1 1]
 1
```

Next, we apply the `all` operation along the rows.

```
--> all(A,2)
ans =
  <int32>  - size: [3 3]

Columns 1 to 3
 1  0  0
 1  0  0
 0  0  1
```

11.3 ANY Any True Function

11.3.1 Usage

Reduces a logical array along a given dimension by testing for any logical 1s. The general syntax for its use is

```
  y = any(x,d)
```

where `x` is an `n`-dimensions array of `logical` type. The output is of `logical` type. The argument `d` is optional, and denotes the dimension along which to operate. The output `y` is the same size as `x`, except that it is singular along the operated direction. So, for example, if `x` is a `3 x 3 x 4` array, and we `any` operation along dimension `d=2`, then the output is of size `3 x 1 x 4`.

11.3.2 Function Internals

The output is computed via

$$y(m_1,\ldots,m_{d-1},1,m_{d+1},\ldots,m_p) = \max_k x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p)$$

If `d` is omitted, then the summation is taken along the first non-singleton dimension of `x`.

11.3.3 Example

The following piece of code demonstrates various uses of the summation function

```
--> A = [1,0,0;1,0,0;0,0,1]
A =
  <int32>  - size: [3 3]
```

```
Columns 1 to 3
 1  0  0
 1  0  0
 0  0  1
```

We start by calling `any` without a dimension argument, in which case it defaults to the first nonsingular dimension (in this case, along the columns or `d = 1`).

```
--> any(A)
ans =
  <logical>  - size: [1 1]
 1
```

Next, we apply the `any` operation along the rows.

```
--> any(A,2)
ans =
  <int32>  - size: [3 3]

Columns 1 to 3
 2  2  2
 2  2  2
 2  2  2
```

11.4 CEIL Ceiling Function

11.4.1 Usage

Computes the ceiling of an n-dimensional array elementwise. The ceiling of a number is defined as the smallest integer that is larger than or equal to that number. The general syntax for its use is

```
   y = ceil(x)
```

where `x` is a multidimensional array of numerical type. The `ceil` function preserves the type of the argument. So integer arguments are not modified, and `float` arrays return `float` arrays as outputs, and similarly for `double` arrays. The `ceil` function is not defined for `complex` or `dcomplex` types.

11.4.2 Example

The following demonstrates the `ceil` function applied to various (numerical) arguments. For integer arguments, the ceil function has no effect:

```
--> ceil(3)
ans =
  <int32>  - size: [1 1]
 3
--> ceil(-3)
ans =
  <int32>  - size: [1 1]
 -3
```

Next, we take the `ceil` of a floating point value:

```
--> ceil(3.023f)
ans =
  <float>  - size: [1 1]
 4
--> ceil(-2.341f)
ans =
  <float>  - size: [1 1]
 -2
```

Note that the return type is a `float` also. Finally, for a `double` type:

```
--> ceil(4.312)
ans =
  <double>  - size: [1 1]
 5
--> ceil(-5.32)
ans =
  <double>  - size: [1 1]
 -5
```

11.5 CONJ Conjugate Function

11.5.1 Usage

Returns the complex conjugate of the input array for all elements. The general syntax for its use is

```
   y = conj(x)
```

where `x` is an `n`-dimensional array of numerical type. The output is the same numerical type as the input. The `conj` function does nothing to real and integer types.

11.5.2 Example

The following demonstrates the complex conjugate applied to a complex scalar.

```
--> conj(3+4*i)
ans =
  <complex>  - size: [1 1]
   3 -4 i
```

The `conj` function has no effect on real arguments:

```
--> conj([2,3,4])
ans =
  <int32>  - size: [1 3]

Columns 1 to 3
 2  3  4
```

For a double-precision complex array,

```
--> conj([2.0+3.0*i,i])
ans =
  <dcomplex>  - size: [1 2]

Columns 1 to 2
   2 -3i    0 -1i
```

11.6 CUMSUM Cumulative Summation Function

11.6.1 Usage

Computes the cumulative sum of an n-dimensional array along a given dimension. The general syntax for its use is

```
  y = cumsum(x,d)
```

where `x` is a multidimensional array of numerical type, and `d` is the dimension along which to perform the cumulative sum. The output `y` is the same size of `x`. Integer types are promoted to `int32`. If the dimension `d` is not specified, then the cumulative sum is applied along the first non-singular dimension.

11.6.2 Function Internals

The output is computed via

$$y(m_1,\ldots,m_{d-1},j,m_{d+1},\ldots,m_p) = \sum_{k=1}^{j} x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p).$$

11.6.3 Example

The default action is to perform the cumulative sum along the first non-singular dimension.

```
--> A = [5,1,3;3,2,1;0,3,1]
A =
  <int32>  - size: [3 3]

Columns 1 to 3
 5  1  3
 3  2  1
 0  3  1
--> cumsum(A)
ans =
  <int32>  - size: [3 3]

Columns 1 to 3
```

```
 5  1  3
 8  3  4
 8  6  5
```

To compute the cumulative sum along the columns:

```
--> cumsum(A,2)
ans =
  <int32>  - size: [3 3]

Columns 1 to 3
 5  6  9
 3  5  6
 0  3  4
```

The cumulative sum also works along arbitrary dimensions

```
--> B(:,:,1) = [5,2;8,9];
--> B(:,:,2) = [1,0;3,0]
B =
  <int32>  - size: [2 2 2]
(:,:,1) =

Columns 1 to 2
 5  2
 8  9
(:,:,2) =

Columns 1 to 2
 1  0
 3  0
--> cumsum(B,3)
ans =
  <int32>  - size: [2 2 2]
(:,:,1) =

Columns 1 to 2
  5   2
  8   9
(:,:,2) =

Columns 1 to 2
  6   2
 11   9
```

11.7 DEAL Multiple Simultaneous Assignments

11.7.1 Usage

When making a function call, it is possible to assign multiple outputs in a single call, (see, e.g., `max` for an example). The `deal` call allows you to do the same thing with a simple assignment. The syntax for its use is

```
[a,b,c,...] = deal(expr)
```

where `expr` is an expression with multiple values. The simplest example is where `expr` is the dereference of a cell array, e.g. `expr <-- A{:}`. In this case, the `deal` call is equivalent to

```
a = A{1}; b = A{2}; C = A{3};
```

Other expressions which are multivalued are structure arrays with multiple entries (non-scalar), where field dereferencing has been applied.

11.8 FLOOR Floor Function

11.8.1 Usage

Computes the floor of an n-dimensional array elementwise. The floor of a number is defined as the smallest integer that is less than or equal to that number. The general syntax for its use is

```
y = floor(x)
```

where `x` is a multidimensional array of numerical type. The `floor` function preserves the type of the argument. So integer arguments are not modified, and `float` arrays return `float` arrays as outputs, and similarly for `double` arrays. The `floor` function is not defined for `complex` or `dcomplex` types.

11.8.2 Example

The following demonstrates the `floor` function applied to various (numerical) arguments. For integer arguments, the floor function has no effect:

```
--> floor(3)
ans =
  <int32>  - size: [1 1]
 3
--> floor(-3)
ans =
  <int32>  - size: [1 1]
 -3
```

Next, we take the `floor` of a floating point value:

```
--> floor(3.023f)
ans =
  <float>  - size: [1 1]
 3
--> floor(-2.341f)
ans =
  <float>  - size: [1 1]
 -3
```

Note that the return type is a `float` also. Finally, for a `double` type:

```
--> floor(4.312)
ans =
  <double>  - size: [1 1]
 4
--> floor(-5.32)
ans =
  <double>  - size: [1 1]
 -6
```

11.9 GETFIELD Get Field Contents

11.9.1 Usage

Given a structure or structure array, returns the contents of the specified field. The first version is for scalar structures, and has the following syntax

```
   y = getfield(x,'fieldname')
```

and is equivalent to `y = x.fieldname` where `x` is a scalar (1 x 1) structure. If `x` is not a scalar structure, then `y` is the first value, i.e., it is equivalent to `y = x(1).fieldname`. The second form allows you to specify a subindex into a structure array, and has the following syntax

```
    y = getfield(x, {m,n}, 'fieldname')
```

and is equivalent to `y = x(m,n).fieldname`. You can chain multiple references together using this syntax.

11.10 IMAG Imaginary Function

11.10.1 Usage

Returns the imaginary part of the input array for all elements. The general syntax for its use is

```
   y = imag(x)
```

where `x` is an `n`-dimensional array of numerical type. The output is the same numerical type as the input, unless the input is `complex` or `dcomplex`. For `complex` inputs, the imaginary part is a floating point array, so that the return type is `float`. For `dcomplex` inputs, the imaginary part is a double precision floating point array, so that the return type is `double`. The `imag` function returns zeros for real and integer types.

11.10.2 Example

The following demonstrates `imag` applied to a complex scalar.

```
--> imag(3+4*i)
ans =
  <float>  - size: [1 1]
 4
```

The imaginary part of real and integer arguments is a vector of zeros, the same type and size of the argument.

```
--> imag([2,4,5,6])
ans =
  <int32>  - size: [1 4]

Columns 1 to 4
 0  0  0  0
```

For a double-precision complex array,

```
--> imag([2.0+3.0*i,i])
ans =
  <double>  - size: [1 2]

Columns 1 to 2
 3  1
```

11.11 MAX Maximum Function

11.11.1 Usage

Computes the maximum of an array along a given dimension, or alternately, computes two arrays (entry-wise) and keeps the smaller value for each array. As a result, the `max` function has a number of syntaxes. The first one computes the maximum of an array along a given dimension. The first general syntax for its use is either

```
   [y,n] = max(x,[],d)
```

where `x` is a multidimensional array of numerical type, in which case the output `y` is the maximum of `x` along dimension `d`. The second argument `n` is the index that results in the maximum. In the event that multiple maxima are present with the same value, the index of the first maximum is used. The second general syntax for the use of the `max` function is

```
[y,n] = max(x)
```

In this case, the maximum is taken along the first non-singleton dimension of `x`. For complex data types, the maximum is based on the magnitude of the numbers. NaNs are ignored in the calculations. The third general syntax for the use of the `max` function is as a comparison function for pairs of arrays. Here, the general syntax is

```
y = max(x,z)
```

where `x` and `z` are either both numerical arrays of the same dimensions, or one of the two is a scalar. In the first case, the output is the same size as both arrays, and is defined elementwise by the smaller of the two arrays. In the second case, the output is defined elementwise by the smaller of the array entries and the scalar.

11.11.2 Function Internals

In the general version of the `max` function which is applied to a single array (using the `max(x,[],d)` or `max(x)` syntaxes), The output is computed via

$$y(m_1,\ldots,m_{d-1},1,m_{d+1},\ldots,m_p) = \max_k x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p),$$

and the output array `n` of indices is calculated via

$$n(m_1,\ldots,m_{d-1},1,m_{d+1},\ldots,m_p) = \arg\max_k x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p)$$

In the two-array version (`max(x,z)`), the single output is computed as

$$y(m_1,\ldots,m_{d-1},1,m_{d+1},\ldots,m_p) = \begin{cases} x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p) & x(\cdots) \leq z(\cdots) \\ z(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p) & z(\cdots) < x(\cdots). \end{cases}$$

11.11.3 Example

The following piece of code demonstrates various uses of the maximum function. We start with the one-array version.

```
--> A = [5,1,3;3,2,1;0,3,1]
A =
  <int32>  - size: [3 3]

Columns 1 to 3
 5  1  3
 3  2  1
 0  3  1
```

We first take the maximum along the columns, resulting in a row vector.

```
--> max(A)
ans =
  <int32>  - size: [1 3]

Columns 1 to 3
 5  3  3
```

Next, we take the maximum along the rows, resulting in a column vector.

```
--> max(A,[],2)
ans =
  <int32>  - size: [3 1]

Columns 1 to 1
 5
 3
 3
```

When the dimension argument is not supplied, `max` acts along the first non-singular dimension. For a row vector, this is the column direction:

```
--> max([5,3,2,9])
ans =
  <int32>  - size: [1 1]
 9
```

For the two-argument version, we can compute the smaller of two arrays, as in this example:

```
--> a = int8(100*randn(4))
a =
  <int8>  - size: [4 4]

Columns 1 to 4
    0   115    15   -20
  -26   127     1   -41
  -12     5   -84    52
   85  -108    -7  -100
--> b = int8(100*randn(4))
b =
  <int8>  - size: [4 4]

Columns 1 to 4
 -30   14  -33  -69
 -62  -71   48    8
 -52    2  -95   75
  40   44  120   -4
--> max(a,b)
ans =
```

```
  <int8>  - size: [4 4]

Columns 1 to 4
   0  115   15  -20
 -26  127   48    8
 -12    5  -84   75
  85   44  120   -4
```

Or alternately, we can compare an array with a scalar

```
--> a = randn(2)
a =
  <double>  - size: [2 2]

Columns 1 to 2
  2.2821506696980594  -0.9318376497645634
 -0.3667213058894895   0.5528981016049694
--> max(a,0)
ans =
  <double>  - size: [2 2]

Columns 1 to 2
 2.2821506696980594  0.0000000000000000
 0.0000000000000000  0.5528981016049694
```

11.12 MEAN Mean Function

11.12.1 Usage

Computes the mean of an array along a given dimension. The general syntax for its use is

```
  y = mean(x,d)
```

where `x` is an `n`-dimensions array of numerical type. The output is of the same numerical type as the input. The argument `d` is optional, and denotes the dimension along which to take the mean. The output `y` is the same size as `x`, except that it is singular along the mean direction. So, for example, if `x` is a `3 x 3 x 4` array, and we compute the mean along dimension `d=2`, then the output is of size `3 x 1 x 4`.

11.12.2 Function Internals

The output is computed via

$$y(m_1,\ldots,m_{d-1},1,m_{d+1},\ldots,m_p) = \frac{1}{N}\sum_{k=1}^{N} x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p)$$

If `d` is omitted, then the mean is taken along the first non-singleton dimension of `x`.

11.12.3 Example

The following piece of code demonstrates various uses of the mean function

```
--> A = [5,1,3;3,2,1;0,3,1]
A =
  <int32>  - size: [3 3]

Columns 1 to 3
 5  1  3
 3  2  1
 0  3  1
```

We start by calling `mean` without a dimension argument, in which case it defaults to the first nonsingular dimension (in this case, along the columns or `d = 1`).

```
--> mean(A)
ans =
  <double>  - size: [1 3]

Columns 1 to 3
 2.6666666666666665  2.0000000000000000  1.6666666666666667
```

Next, we take the mean along the rows.

```
--> mean(A,2)
ans =
  <double>  - size: [3 1]

Columns 1 to 1
 3.0000000000000000
 2.0000000000000000
 1.3333333333333333
```

11.13 MIN Minimum Function

11.13.1 Usage

Computes the minimum of an array along a given dimension, or alternately, computes two arrays (entry-wise) and keeps the smaller value for each array. As a result, the `min` function has a number of syntaxes. The first one computes the minimum of an array along a given dimension. The first general syntax for its use is either

```
[y,n] = min(x,[],d)
```

where `x` is a multidimensional array of numerical type, in which case the output `y` is the minimum of `x` along dimension `d`. The second argument `n` is the index that results in the minimum. In the event that multiple minima are present with the same value, the index of the first minimum is used. The second general syntax for the use of the `min` function is

```
[y,n] = min(x)
```

In this case, the minimum is taken along the first non-singleton dimension of `x`. For complex data types, the minimum is based on the magnitude of the numbers. NaNs are ignored in the calculations. The third general syntax for the use of the `min` function is as a comparison function for pairs of arrays. Here, the general syntax is

```
y = min(x,z)
```

where `x` and `z` are either both numerical arrays of the same dimensions, or one of the two is a scalar. In the first case, the output is the same size as both arrays, and is defined elementwise by the smaller of the two arrays. In the second case, the output is defined elementwise by the smaller of the array entries and the scalar.

11.13.2 Function Internals

In the general version of the `min` function which is applied to a single array (using the `min(x,[],d)` or `min(x)` syntaxes), The output is computed via

$$y(m_1,\ldots,m_{d-1},1,m_{d+1},\ldots,m_p) = \min_k x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p),$$

and the output array `n` of indices is calculated via

$$n(m_1,\ldots,m_{d-1},1,m_{d+1},\ldots,m_p) = \arg\min_k x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p)$$

In the two-array version (`min(x,z)`), the single output is computed as

$$y(m_1,\ldots,m_{d-1},1,m_{d+1},\ldots,m_p) = \begin{cases} x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p) & x(\cdots) \leq z(\cdots) \\ z(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p) & z(\cdots) < x(\cdots). \end{cases}$$

11.13.3 Example

The following piece of code demonstrates various uses of the minimum function. We start with the one-array version.

```
--> A = [5,1,3;3,2,1;0,3,1]
A =
  <int32>  - size: [3 3]

Columns 1 to 3
 5  1  3
 3  2  1
 0  3  1
```

We first take the minimum along the columns, resulting in a row vector.

```
--> min(A)
ans =
  <int32>  - size: [1 3]

Columns 1 to 3
 0  1  1
```

Next, we take the minimum along the rows, resulting in a column vector.

```
--> min(A,[],2)
ans =
  <int32>  - size: [3 1]

Columns 1 to 1
 1
 1
 0
```

When the dimension argument is not supplied, `min` acts along the first non-singular dimension. For a row vector, this is the column direction:

```
--> min([5,3,2,9])
ans =
  <int32>  - size: [1 1]
 2
```

For the two-argument version, we can compute the smaller of two arrays, as in this example:

```
--> a = int8(100*randn(4))
a =
  <int8>  - size: [4 4]

Columns 1 to 4
  -3   59   -5  110
 -14   70  -16   -3
  69  -93    1  118
 -23    0   16  -74
--> b = int8(100*randn(4))
b =
  <int8>  - size: [4 4]

Columns 1 to 4
   64   -51    74    84
  -40   -62   -84  -126
 -102   -12    43   -54
   69    50   -56    29
--> min(a,b)
ans =
```

```
  <int8>  - size: [4 4]

Columns 1 to 4
   -3   -51    -5    84
  -40   -62   -84  -126
 -102   -93     1   -54
  -23     0   -56   -74
```

Or alternately, we can compare an array with a scalar

```
--> a = randn(2)
a =
  <double>  - size: [2 2]

Columns 1 to 2
 -0.8511880774404023  -0.6258277325427632
  0.8414718191173367   1.3391173182906131
--> min(a,0)
ans =
  <double>  - size: [2 2]

Columns 1 to 2
 -0.8511880774404023  -0.6258277325427632
  0.0000000000000000   0.0000000000000000
```

11.14 PROD Product Function

11.14.1 Usage

Computes the product of an array along a given dimension. The general syntax for its use is

```
   y = prod(x,d)
```

where `x` is an `n`-dimensions array of numerical type. The output is of the same numerical type as the input, except for integer types, which are automatically promoted to `int32`. The argument `d` is optional, and denotes the dimension along which to take the product. The output is computed via

$$y(m_1,\ldots,m_{d-1},1,m_{d+1},\ldots,m_p) = \prod_k x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p)$$

If `d` is omitted, then the product is taken along the first non-singleton dimension of `x`.

11.14.2 Example

The following piece of code demonstrates various uses of the product function

```
--> A = [5,1,3;3,2,1;0,3,1]
A =
```

```
  <int32>  - size: [3 3]

Columns 1 to 3
 5  1  3
 3  2  1
 0  3  1
```

We start by calling `prod` without a dimension argument, in which case it defaults to the first nonsingular dimension (in this case, along the columns or `d = 1`).

```
--> prod(A)
ans =
  <int32>  - size: [1 3]

Columns 1 to 3
 0  6  3
```

Next, we take the product along the rows.

```
--> prod(A,2)
ans =
  <int32>  - size: [3 1]

Columns 1 to 1
 15
  6
  0
```

11.15 REAL Real Function

11.15.1 Usage

Returns the real part of the input array for all elements. The general syntax for its use is

```
   y = real(x)
```

where `x` is an `n`-dimensional array of numerical type. The output is the same numerical type as the input, unless the input is `complex` or `dcomplex`. For `complex` inputs, the real part is a floating point array, so that the return type is `float`. For `dcomplex` inputs, the real part is a double precision floating point array, so that the return type is `double`. The `real` function does nothing to real and integer types.

11.15.2 Example

The following demonstrates the `real` applied to a complex scalar.

```
--> real(3+4*i)
ans =
  <float>  - size: [1 1]
 3
```

The `real` function has no effect on real arguments:

```
--> real([2,3,4])
ans =
  <int32>  - size: [1 3]

Columns 1 to 3
 2  3  4
```

For a double-precision complex array,

```
--> real([2.0+3.0*i,i])
ans =
  <double>  - size: [1 2]

Columns 1 to 2
 2  0
```

11.16 ROUND Round Function

11.16.1 Usage

Rounds an n-dimensional array to the nearest integer elementwise. The general syntax for its use is

```
   y = round(x)
```

where `x` is a multidimensional array of numerical type. The `round` function preserves the type of the argument. So integer arguments are not modified, and `float` arrays return `float` arrays as outputs, and similarly for `double` arrays. The `round` function is not defined for `complex` or `dcomplex` types.

11.16.2 Example

The following demonstrates the `round` function applied to various (numerical) arguments. For integer arguments, the round function has no effect:

```
--> round(3)
ans =
  <int32>  - size: [1 1]
 3
--> round(-3)
ans =
  <int32>  - size: [1 1]
 -3
```

Next, we take the `round` of a floating point value:

```
--> round(3.023f)
ans =
```

```
  <float>  - size: [1 1]
 3
--> round(-2.341f)
ans =
  <float>  - size: [1 1]
 -2
```

Note that the return type is a `float` also. Finally, for a `double` type:

```
--> round(4.312)
ans =
  <double>  - size: [1 1]
 4
--> round(-5.32)
ans =
  <double>  - size: [1 1]
 -5
```

11.17 STD Standard Deviation Function

11.17.1 Usage

Computes the standard deviation of an array along a given dimension. The general syntax for its use is

```
  y = std(x,d)
```

where `x` is an `n`-dimensions array of numerical type. The output is of the same numerical type as the input. The argument `d` is optional, and denotes the dimension along which to take the variance. The output `y` is the same size as `x`, except that it is singular along the mean direction. So, for example, if `x` is a `3 x 3 x 4` array, and we compute the mean along dimension `d=2`, then the output is of size `3 x 1 x 4`.

11.17.2 Example

The following piece of code demonstrates various uses of the `std` function

```
--> A = [5,1,3;3,2,1;0,3,1]
A =
  <int32>  - size: [3 3]

Columns 1 to 3
 5  1  3
 3  2  1
 0  3  1
```

We start by calling `std` without a dimension argument, in which case it defaults to the first nonsingular dimension (in this case, along the columns or `d = 1`).

```
--> std(A)
ans =
  <double>  - size: [1 3]

Columns 1 to 3
 2.5166114784235831  1.0000000000000000  1.1547005383792515
```

Next, we take the variance along the rows.

```
--> std(A,2)
ans =
  <double>  - size: [3 1]

Columns 1 to 1
 2.0000000000000000
 1.0000000000000000
 1.5275252316519468
```

11.18 SUB2IND Convert Multiple Indexing To Linear Indexing

11.18.1 Usage

The `sub2ind` function converts a multi-dimensional indexing expression into a linear (or vector) indexing expression. The syntax for its use is

```
   y = sub2ind(sizevec,d1,d2,...,dn)
```

where `sizevec` is the size of the array being indexed into, and each `di` is a vector of the same length, containing index values. The basic idea behind `sub2ind` is that it makes

```
  [z(d1(1),d2(1),...,dn(1)),...,z(d1(n),d2(n),...,dn(n))]
```

equivalent to

```
  z(sub2ind(size(z),d1,d2,...,dn))
```

where the later form is using vector indexing, and the former one is using native, multi-dimensional indexing.

11.18.2 Example

Suppose we have a simple `3 x 4` matrix `A` containing some random integer elements

```
--> A = randi(ones(3,4),10*ones(3,4))
A =
  <int32>  - size: [3 4]
```

```
Columns 1 to 4
  2   2   6   3
  2  10  10   2
  6   1   9   7
```

We can extract the elements `(1,3),(2,3),(3,4)` of `A` via `sub2ind`. To calculate which elements of `A` this corresponds to, we can use `sub2ind` as

```
--> n = sub2ind(size(A),1:3,2:4)
n =
  <int32>  - size: [1 3]

Columns 1 to 3
  4   8  12
--> A(n)
ans =
  <int32>  - size: [1 3]

Columns 1 to 3
  2  10   7
```

11.19 SUM Sum Function

11.19.1 Usage

Computes the summation of an array along a given dimension. The general syntax for its use is

```
  y = sum(x,d)
```

where `x` is an `n`-dimensions array of numerical type. The output is of the same numerical type as the input. The argument `d` is optional, and denotes the dimension along which to take the summation. The output `y` is the same size as `x`, except that it is singular along the summation direction. So, for example, if `x` is a `3 x 3 x 4` array, and we compute the summation along dimension `d=2`, then the output is of size `3 x 1 x 4`.

11.19.2 Function Internals

The output is computed via

$$y(m_1,\ldots,m_{d-1},1,m_{d+1},\ldots,m_p) = \sum_{k} x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p)$$

If `d` is omitted, then the summation is taken along the first non-singleton dimension of `x`.

11.19.3 Example

The following piece of code demonstrates various uses of the summation function

```
--> A = [5,1,3;3,2,1;0,3,1]
A =
  <int32>  - size: [3 3]

Columns 1 to 3
 5  1  3
 3  2  1
 0  3  1
```

We start by calling `sum` without a dimension argument, in which case it defaults to the first nonsingular dimension (in this case, along the columns or `d = 1`).

```
--> sum(A)
ans =
  <int32>  - size: [1 3]

Columns 1 to 3
 8  6  5
```

Next, we take the sum along the rows.

```
--> sum(A,2)
ans =
  <int32>  - size: [3 1]

Columns 1 to 1
 9
 6
 4
```

11.20 VAR Variance Function

11.20.1 Usage

Computes the variance of an array along a given dimension. The general syntax for its use is

```
  y = var(x,d)
```

where `x` is an `n`-dimensions array of numerical type. The output is of the same numerical type as the input. The argument `d` is optional, and denotes the dimension along which to take the variance. The output `y` is the same size as `x`, except that it is singular along the mean direction. So, for example, if `x` is a `3 x 3 x 4` array, and we compute the mean along dimension `d=2`, then the output is of size `3 x 1 x 4`.

11.20.2 Function Internals

The output is computed via

$$y(m_1,\ldots,m_{d-1},1,m_{d+1},\ldots,m_p) = \frac{1}{N-1}\sum_{k=1}^{N}\left(x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p) - \bar{x}\right)^2,$$

where

$$\bar{x} = \frac{1}{N}\sum_{k=1}^{N} x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p)$$

If `d` is omitted, then the mean is taken along the first non-singleton dimension of `x`.

11.20.3 Example

The following piece of code demonstrates various uses of the var function

```
--> A = [5,1,3;3,2,1;0,3,1]
A =
  <int32>  - size: [3 3]

Columns 1 to 3
 5  1  3
 3  2  1
 0  3  1
```

We start by calling `var` without a dimension argument, in which case it defaults to the first nonsingular dimension (in this case, along the columns or `d = 1`).

```
--> var(A)
ans =
  <double>  - size: [1 3]

Columns 1 to 3
 6.3333333333333330  1.0000000000000000  1.3333333333333333
```

Next, we take the variance along the rows.

```
--> var(A,2)
ans =
  <double>  - size: [3 1]

Columns 1 to 1
 4.0000000000000000
 1.0000000000000000
 2.3333333333333335
```

11.21 VEC Reshape to a Vector

11.21.1 Usage

Reshapes an n-dimensional array into a column vector. The general syntax for its use is

```
   y = vec(x)
```

where `x` is an n-dimensional array (not necessarily numeric). This function is equivalent to the expression `y = x(:)`.

11.21.2 Example

A simple example of the `vec` operator reshaping a 2D matrix:

```
--> A = [1,2,4,3;2,3,4,5]
A =
  <int32>  - size: [2 4]

Columns 1 to 4
 1  2  4  3
 2  3  4  5
--> vec(A)
ans =
  <int32>  - size: [8 1]

Columns 1 to 1
 1
 2
 2
 3
 4
 4
 3
 5
```

Chapter 12

Inspection Functions

12.1 CLEAR Clear or Delete a Variable

12.1.1 Usage

Clears a set of variables from the current context, or alternately, delete all variables defined in the current context. There are two formats for the function call. The first is the explicit form in which a list of variables are provided:

```
clear a1 a2 ...
```

The variables can be persistent or global, and they will be deleted. The second form

```
clear 'all'
```

clears all variables from the current context. With no arguments, `clear` defaults to clearing `'all'`.

12.1.2 Example

Here is a simple example of using `clear` to delete a variable. First, we create a variable called `a`:

```
--> a = 53
a =
  <int32>  - size: [1 1]
 53
```

Next, we clear `a` using the `clear` function, and verify that it is deleted.

```
--> clear a
--> a
Error: Undefined function or variable a
```

12.2 EXIST Test for Existence

12.2.1 Usage

Tests for the existence of a variable, function, directory or file. The general syntax for its use is

```
y = exist(item,kind)
```

where `item` is a string containing the name of the item to look for, and `kind` is a string indicating the type of the search. The `kind` must be one of

- `'builtin'` checks for built-in functions
- `'dir'` checks for directories
- `'file'` checks for files
- `'var'` checks for variables
- `'all'` checks all possibilities (same as leaving out `kind`)

You can also leave the `kind` specification out, in which case the calling syntax is

```
y = exist(item)
```

The return code is one of the following:

- 0 - if `item` does not exist
- 1 - if `item` is a variable in the workspace
- 2 - if `item` is an M file on the search path, a full pathname to a file, or an ordinary file on your search path
- 5 - if `item` is a built-in FreeMat function
- 7 - if `item` is a directory

Note: previous to version `1.10`, `exist` used a different notion of existence for variables: a variable was said to exist if it was defined and non-empty. This test is now performed by `isset`.

12.2.2 Example

Some examples of the `exist` function. Note that generally `exist` is used in functions to test for keywords. For example,

```
function y = testfunc(a, b, c)
if (~exist('c'))
  % c was not defined, so establish a default
  c = 13;
end
y = a + b + c;
```

An example of `exist` in action.

```
--> a = randn(3,5,2)
a =
  <double>  - size: [3 5 2]
(:,:,1) =

Columns 1 to 3
  0.88871349656451581  -0.27486926931161132  -0.12024249625514018
 -0.90519861711237160   0.26884652521190833   1.90471605852950709
 -1.65189724917687153   0.16892429650998678   0.51341456129698237

Columns 4 to 5
  0.23474524092873861   0.28152354823275738
 -0.05325434204406199  -1.61961083433645769
 -0.57950864136569458   0.78628503285432572
(:,:,2) =

Columns 1 to 3
  0.82461267864970456  -0.58233963047628279  -0.69861236877830091
 -0.50220881813883467   2.43683687159149187   1.26785652397925097
 -0.99659329315144363  -0.55300573033678591  -0.33252184524739037

Columns 4 to 5
  0.35908413820778551  -2.59873004781618944
 -1.47481790392009282  -0.42393856108254580
  2.29840609365120008   0.50244492229832827
--> b = []
b =
  <double>  - size: [0 0]
  []
--> who
  Variable Name      Type   Flags             Size
              a    double                    [3 5 2]
            ans   logical                    [1 1]
              b    double                    [0 0]
              c     int32                    [1 3]
              f    string                    [1 5]
              p    double                    [1 256]
              x      cell                    [2 1]
              y    struct                    [1 1]
--> exist('a')
ans =
  <int32>  - size: [1 1]
 1
--> exist('b')
```

```
ans =
  <int32>  - size: [1 1]
 1
--> exist('c')
ans =
  <int32>  - size: [1 1]
 1
```

12.3 FIELDNAMES Fieldnames of a Structure

12.3.1 Usage

Returns a cell array containing the names of the fields in a structure array. The syntax for its use is

```
x = fieldnames(y)
```

where `y` is a structure array of object array. The result is a cell array, with one entry per field in `y`.

12.3.2 Example

We define a simple structure array:

```
--> y.foo = 3; y.goo = 'hello';
--> x = fieldnames(y)
x =
  <cell array> - size: [2 1]

Columns 1 to 1
 foo
 goo
```

12.4 ISA Test Type of Variable

12.4.1 Usage

Tests the type of a variable. The syntax for its use is

```
y = isa(x,type)
```

where `x` is the variable to test, and `type` is the type. Supported built-in types are

- `'cell'` for cell-arrays
- `'struct'` for structure-arrays
- `'logical'` for logical arrays
- `'uint8'` for unsigned 8-bit integers

- `'int8'` for signed 8-bit integers
- `'uint16'` for unsigned 16-bit integers
- `'int16'` for signed 16-bit integers
- `'uint32'` for unsigned 32-bit integers
- `'int32'` for signed 32-bit integers
- `'float'` for 32-bit floating point numbers
- `'double'` for 64-bit floating point numbers
- `'complex'` for complex floating point numbers with 32-bits per field
- `'dcomplex'` for complex floating point numbers with 64-bits per field
- `'string'` for string arrays

If the argument is a user-defined type (via the `class` function), then the name of that class is returned.

12.4.2 Examples

Here are some examples of the `isa` call.

```
--> a = {1}
a =
  <cell array> - size: [1 1]
 [1]
--> isa(a,'string')
ans =
  <logical>  - size: [1 1]
 0
--> isa(a,'cell')
ans =
  <logical>  - size: [1 1]
 1
```

Here we use `isa` along with shortcut boolean evaluation to safely determine if a variable contains the string `'hello'`

```
--> a = 'hello'
a =
  <string>  - size: [1 5]
 hello
--> isa(a,'string') && strcmp(a,'hello')
ans =
  <logical>  - size: [1 1]
 1
```

12.5 ISEMPTY Test For Variable Empty

12.5.1 Usage

The `isempty` function returns a boolean that indicates if the argument variable is empty or not. The general syntax for its use is

```
  y = isempty(x).
```

12.5.2 Examples

Here are some examples of the `isempty` function

```
--> a = []
a =
  <double>  - size: [0 0]
  []
--> isempty(a)
ans =
  <logical>  - size: [1 1]
 1
--> b = 1:3
b =
  <int32>  - size: [1 3]

Columns 1 to 3
 1  2  3
--> isempty(b)
ans =
  <logical>  - size: [1 1]
 0
```

Note that if the variable is not defined, `isempty` does not return true.

```
--> isempty(x)
ans =
  <logical>  - size: [1 1]
 0
```

12.6 ISFIELD Test for Existence of a Structure Field

12.6.1 Usage

Given a structure array, tests to see if that structure array contains a field with the given name. The syntax for its use is

```
  y = isfield(x,field)
```

and returns a logical `1` if `x` has a field with the name `field` and a logical `0` if not. It also returns a logical `0` if the argument `x` is not a structure array.

12.6.2 Example

Here we define a simple struct, and then test for some fields

```
--> a.foo = 32
a =
  <structure array> - size: [1 1]
    foo: [32]
--> a.goo = 64
a =
  <structure array> - size: [1 1]
    foo: [32]
    goo: [64]
--> isfield(a,'goo')
ans =
  <logical>  - size: [1 1]
 1
--> isfield(a,'got')
ans =
  <logical>  - size: [1 1]
 0
--> isfield(pi,'round')
ans =
  <logical>  - size: [1 1]
 0
```

12.7 ISHANDLE Test for Graphics Handle

12.7.1 Usage

Given a constant, this routine will test to see if the constant is a valid graphics handle or not. The syntax for its use is

```
  y = ishandle(h,type)
```

and returns a logical `1` if `x` is a handle of type `type` and a logical `0` if not.

12.8 ISINF Test for infinities

12.8.1 Usage

Returns true for entries of an array that are infs (i.e., infinities). The usage is

```
   y = isinf(x)
```

The result is a logical array of the same size as `x`, which is true if `x` is not-a-number, and false otherwise. Note that for `complex` or `dcomplex` data types that the result is true if either the real or imaginary parts are infinite.

12.8.2 Example

Suppose we have an array of floats with one element that is `inf`:

```
--> a = [1.2 3.4 inf 5]
a =
  <double>  - size: [1 4]

Columns 1 to 4
   1.2    3.4    inf    5.0
--> isinf(a)
ans =
  <logical>  - size: [1 4]

Columns 1 to 4
 0  0  1  0
--> b = 3./[2 5 0 3 1]
b =
  <double>  - size: [1 5]

Columns 1 to 5
   1.5    0.6    inf    1.0    3.0
```

12.9 ISNAN Test for Not-a-Numbers

12.9.1 Usage

Returns true for entries of an array that are NaN's (i.e., Not-a-Numbers). The usage is

```
   y = isnan(x)
```

The result is a logical array of the same size as `x`, which is true if `x` is not-a-number, and false otherwise. Note that for `complex` or `dcomplex` data types that the result is true if either the real or imaginary parts are NaNs.

12.9.2 Example

Suppose we have an array of floats with one element that is `nan`:

```
--> a = [1.2 3.4 nan 5]
a =
  <double>  - size: [1 4]

Columns 1 to 4
   1.2    3.4    nan    5.0
--> isnan(a)
ans =
  <logical>  - size: [1 4]
```

```
Columns 1 to 4
 0  0  1  0
```

12.10 ISSET Test If Variable Set

12.10.1 Usage

Tests for the existence and non-emptiness of a variable. the general syntax for its use is

```
y = isset('name')
```

where `name` is the name of the variable to test. This is functionally equivalent to

```
y = exist('name','var') & ~isempty(name)
```

It returns a `logical` 1 if the variable is defined in the current workspace, and is not empty, and returns a 0 otherwise.

12.10.2 Example

Some simple examples of using `isset`

```
--> who
  Variable Name      Type   Flags             Size
              a    double                    [23 12 5]
            ans    uint32                    [1 1]
              c     int32                    [1 3]
              f    string                    [1 5]
              p    double                    [1 256]
              x      cell                    [2 1]
              y    struct                    [1 1]
--> isset('a')
ans =
  <logical>  - size: [1 1]
 1
--> a = [];
--> isset('a')
ans =
  <logical>  - size: [1 1]
 0
--> a = 2;
--> isset('a')
ans =
  <logical>  - size: [1 1]
 1
```

12.11 ISSPARSE Test for Sparse Matrix

12.11.1 Usage

Test a matrix to see if it is sparse or not. The general format for its use is

```
y = issparse(x)
```

This function returns true if `x` is encoded as a sparse matrix, and false otherwise.

12.11.2 Example

Here is an example of using `issparse`:

```
--> a = [1,0,0,5;0,3,2,0]
a =
  <int32>  - size: [2 4]

Columns 1 to 4
 1  0  0  5
 0  3  2  0
--> issparse(a)
ans =
  <logical>  - size: [1 1]
 0
--> A = sparse(a)
A =
  <int32>  - size: [2 4]
Matrix is sparse with 4 nonzeros
--> issparse(A)
ans =
  <logical>  - size: [1 1]
 1
```

12.12 SIZE Size of a Variable

12.12.1 Usage

Returns the size of a variable. There are two syntaxes for its use. The first syntax returns the size of the array as a vector of integers, one integer for each dimension

```
[d1,d2,...,dn] = size(x)
```

The other format returns the size of `x` along a particular dimension:

```
d = size(x,n)
```

where `n` is the dimension along which to return the size.

12.12.2 Example

```
--> a = randn(23,12,5);
--> size(a)
ans =
  <uint32>  - size: [1 3]

Columns 1 to 3
 23  12   5
```

Here is an example of the second form of `size`.

```
--> size(a,2)
ans =
  <uint32>  - size: [1 1]
 12
```

12.13 WHERE Get Information on Program Stack

12.13.1 Usage

Returns information on the current stack. The usage is

```
   where
```

The result is a kind of stack trace that indicates the state of the current call stack, and where you are relative to the stack.

12.13.2 Example

Suppose we have the following chain of functions.

```
     chain1.m
function chain1
  a = 32;
  b = a + 5;
  chain2(b)

     chain2.m
function chain2(d)
  d = d + 5;
  chain3

     chain3.m
function chain3
  g = 54;
  f = g + 1;
  keyboard
```

The execution of the `where` command shows the stack trace.

```
--> chain1
[chain3,4] --> where
In base(base), line 0, column 0
In Eval(chain1), line 1, column 7
In chain1(chain1), line 4, column 9
In chain2(chain2), line 3, column 9
In chain3(chain3), line 4, column 11
In Eval(where), line 1, column 6
In where(built in), line 0, column 0
```

12.14 WHICH Get Information on Function

12.14.1 Usage

Returns information on a function (if defined). The usage is

```
which(fname)
```

where `fname` is a `string` argument that contains the name of the function. For functions and scripts defined via `.m` files, the `which` command returns the location of the source file:

```
y = which(fname)
```

will return the filename for the `.m` file corresponding to the given function, and an empty string otherwise.

12.14.2 Example

First, we apply the `which` command to a built in function.

```
--> which fft
Function fft is a built in function
```

Next, we apply it to a function defined via a `.m` file.

```
--> which fliplr
Function fliplr, M-File function in file '/home/basu/Dev/trunk/FreeMat2/MFiles/fliplr.m'
```

12.15 WHO Describe Currently Defined Variables

12.15.1 Usage

Reports information on either all variables in the current context or on a specified set of variables. For each variable, the `who` function indicates the size and type of the variable as well as if it is a global or persistent. There are two formats for the function call. The first is the explicit form, in which a list of variables are provided:

```
who a1 a2 ...
```

In the second form

```
who
```

the `who` function lists all variables defined in the current context (as well as global and persistent variables). Note that there are two alternate forms for calling the `who` function:

```
who 'a1' 'a2' ...
```

and

```
who('a1','a2',...)
```

12.15.2 Example

Here is an example of the general use of `who`, which lists all of the variables defined.

```
--> c = [1,2,3];
--> f = 'hello';
--> p = randn(1,256);
--> who
  Variable Name      Type   Flags             Size
              c     int32                    [1 3]
              f    string                    [1 5]
              p    double                    [1 256]
```

In the second case, we examine only a specific variable:

```
--> who c
  Variable Name      Type   Flags             Size
              c     int32                    [1 3]
--> who('c')
  Variable Name      Type   Flags             Size
              c     int32                    [1 3]
```

Chapter 13

Type Cast Functions

13.1 COMPLEX Convert to 32-bit Complex Floating Point

13.1.1 Usage

Converts the argument to a 32-bit complex floating point number. The syntax for its use is

```
y = complex(x)
```

where `x` is an `n`-dimensional numerical array. Conversion follows the general C rules. Note that both `NaN` and `Inf` in the real and imaginary parts are both preserved under type conversion.

13.1.2 Example

The following piece of code demonstrates several uses of `complex`. First, we convert from an integer (the argument is an integer because no decimal is present):

```
--> complex(200)
ans =
  <complex>  - size: [1 1]
  200   0 i
```

In the next example, a double precision argument is passed in (the presence of a decimal without the `f` suffix implies double precision).

```
--> complex(400.0)
ans =
  <complex>  - size: [1 1]
  400   0 i
```

In the next example, a dcomplex argument is passed in.

```
--> complex(3.0+4.0*i)
ans =
  <complex>  - size: [1 1]
  3+ 4 i
```

In the next example, a string argument is passed in. The string argument is converted into an integer array corresponding to the ASCII values of each character.

```
--> complex('he')
ans =
  <complex>  - size: [1 2]

Columns 1 to 2
  104   0 i   101   0 i
```

In the next example, the NaN argument is converted.

```
--> complex(nan)
ans =
  <complex>  - size: [1 1]
                                    nan                                       0 i
```

In the last example, a cell-array is passed in. For cell-arrays and structure arrays, the result is an error.

```
--> complex({4})
Error: Cannot convert cell-arrays to any other type.
```

13.2 DCOMPLEX Convert to 32-bit Complex Floating Point

13.2.1 Usage

Converts the argument to a 32-bit complex floating point number. The syntax for its use is

```
   y = dcomplex(x)
```

where `x` is an `n`-dimensional numerical array. Conversion follows the general C rules. Note that both `NaN` and `Inf` in the real and imaginary parts are both preserved under type conversion.

13.2.2 Example

The following piece of code demonstrates several uses of `dcomplex`. First, we convert from an integer (the argument is an integer because no decimal is present):

```
--> dcomplex(200)
ans =
  <dcomplex>  - size: [1 1]
  200   0i
```

In the next example, a double precision argument is passed in (the presence of a decimal without the `f` suffix implies double precision).

```
--> dcomplex(400.0)
ans =
  <dcomplex>  - size: [1 1]
  400   0i
```

In the next example, a complex argument is passed in.

```
--> dcomplex(3.0+4.0*i)
ans =
  <dcomplex>  - size: [1 1]
  3+ 4i
```

In the next example, a string argument is passed in. The string argument is converted into an integer array corresponding to the ASCII values of each character.

```
--> dcomplex('h')
ans =
  <dcomplex>  - size: [1 1]
  104   0i
```

In the next example, the `NaN` argument is converted.

```
--> dcomplex(nan)
ans =
  <dcomplex>  - size: [1 1]
                                    nan                                        0i
```

In the last example, a cell-array is passed in. For cell-arrays and structure arrays, the result is an error.

```
--> dcomplex({4})
Error: Cannot convert cell-arrays to any other type.
```

13.3 DOUBLE Convert to 64-bit Floating Point

13.3.1 Usage

Converts the argument to a 64-bit floating point number. The syntax for its use is

```
   y = double(x)
```

where `x` is an `n`-dimensional numerical array. Conversion follows the general C rules. Note that both `NaN` and `Inf` are both preserved under type conversion.

13.3.2 Example

The following piece of code demonstrates several uses of `double`. First, we convert from an integer (the argument is an integer because no decimal is present):

```
--> double(200)
ans =
  <double>  - size: [1 1]
 200
```

In the next example, a single precision argument is passed in (the presence of the `f` suffix implies single precision).

```
--> double(400.0f)
ans =
  <double>  - size: [1 1]
 400
```

In the next example, a dcomplex argument is passed in. The result is the real part of the argument, and in this context, `double` is equivalent to the function `real`.

```
--> double(3.0+4.0*i)
ans =
  <double>  - size: [1 1]
 3
```

In the next example, a string argument is passed in. The string argument is converted into an integer array corresponding to the ASCII values of each character.

```
--> double('helo')
ans =
  <double>  - size: [1 4]

Columns 1 to 4
 104  101  108  111
```

In the last example, a cell-array is passed in. For cell-arrays and structure arrays, the result is an error.

```
--> double({4})
Error: Cannot convert cell-arrays to any other type.
```

13.4 FLOAT Convert to 32-bit Floating Point

13.4.1 Usage

Converts the argument to a 32-bit floating point number. The syntax for its use is

```
   y = float(x)
```

where `x` is an `n`-dimensional numerical array. Conversion follows the general C rules. Note that both `NaN` and `Inf` are both preserved under type conversion.

13.4.2 Example

The following piece of code demonstrates several uses of `float`. First, we convert from an integer (the argument is an integer because no decimal is present):

```
--> float(200)
ans =
  <float>  - size: [1 1]
 200
```

In the next example, a double precision argument is passed in (the presence of a decimal without the `f` suffix implies double precision).

```
--> float(400.0)
ans =
  <float>  - size: [1 1]
 400
```

In the next example, a dcomplex argument is passed in. The result is the real part of the argument, and in this context, `float` is equivalent to the function `real`.

```
--> float(3.0+4.0*i)
ans =
  <float>  - size: [1 1]
 3
```

In the next example, a string argument is passed in. The string argument is converted into an integer array corresponding to the ASCII values of each character.

```
--> float('helo')
ans =
  <float>  - size: [1 4]

Columns 1 to 4
 104  101  108  111
```

In the last example, a cell-array is passed in. For cell-arrays and structure arrays, the result is an error.

```
--> float({4})
Error: Cannot convert cell-arrays to any other type.
```

13.5 INT16 Convert to Signed 16-bit Integer

13.5.1 Usage

Converts the argument to an signed 16-bit Integer. The syntax for its use is

```
   y = int16(x)
```

where `x` is an `n`-dimensional numerical array. Conversion follows the general C rules (e.g., if `x` is outside the normal range for a signed 16-bit integer of `[-32768,32767]`, the least significant 16 bits of `x` are used after conversion to a signed integer). Note that both `NaN` and `Inf` both map to 0.

13.5.2 Example

The following piece of code demonstrates several uses of `int16`. First, the routine uses

```
--> int16(100)
ans =
  <int16>  - size: [1 1]
 100
--> int16(-100)
ans =
  <int16>  - size: [1 1]
 -100
```

In the next example, an integer outside the range of the type is passed in. The result is the 16 least significant bits of the argument.

```
--> int16(40000)
ans =
  <int16>  - size: [1 1]
 -25536
```

In the next example, a positive double precision argument is passed in. The result is the signed integer that is closest to the argument.

```
--> int16(pi)
ans =
  <int16>  - size: [1 1]
 3
```

In the next example, a complex argument is passed in. The result is the signed integer that is closest to the real part of the argument.

```
--> int16(5+2*i)
ans =
  <int16>  - size: [1 1]
 5
```

In the next example, a string argument is passed in. The string argument is converted into an integer array corresponding to the ASCII values of each character.

```
--> int16('helo')
ans =
  <int16>  - size: [1 4]

Columns 1 to 4
 104  101  108  111
```

In the last example, a cell-array is passed in. For cell-arrays and structure arrays, the result is an error.

```
--> int16({4})
Error: Cannot convert cell-arrays to any other type.
```

13.6 INT32 Convert to Signed 32-bit Integer

13.6.1 Usage

Converts the argument to an signed 32-bit Integer. The syntax for its use is

```
y = int32(x)
```

where `x` is an `n`-dimensional numerical array. Conversion follows the general C rules (e.g., if `x` is outside the normal range for a signed 32-bit integer of `[-2147483648,2147483647]`, the least significant 32 bits of `x` are used after conversion to a signed integer). Note that both `NaN` and `Inf` both map to 0.

13.6.2 Example

The following piece of code demonstrates several uses of `int32`. First, the routine uses

```
--> int32(100)
ans =
  <int32>  - size: [1 1]
 100
--> int32(-100)
ans =
  <int32>  - size: [1 1]
 -100
```

In the next example, an integer outside the range of the type is passed in. The result is the 32 least significant bits of the argument.

```
--> int32(40e9)
ans =
  <int32>  - size: [1 1]
 -2147483648
```

In the next example, a positive double precision argument is passed in. The result is the signed integer that is closest to the argument.

```
--> int32(pi)
ans =
  <int32>  - size: [1 1]
 3
```

In the next example, a complex argument is passed in. The result is the signed integer that is closest to the real part of the argument.

```
--> int32(5+2*i)
ans =
  <int32>  - size: [1 1]
 5
```

In the next example, a string argument is passed in. The string argument is converted into an integer array corresponding to the ASCII values of each character.

```
--> int32('helo')
ans =
  <int32>  - size: [1 4]

Columns 1 to 4
 104  101  108  111
```

In the last example, a cell-array is passed in. For cell-arrays and structure arrays, the result is an error.

```
--> int32({4})
Error: Cannot convert cell-arrays to any other type.
```

13.7 INT8 Convert to Signed 8-bit Integer

13.7.1 Usage

Converts the argument to an signed 8-bit Integer. The syntax for its use is

```
   y = int8(x)
```

where `x` is an `n`-dimensional numerical array. Conversion follows the general C rules (e.g., if `x` is outside the normal range for a signed 8-bit integer of `[-128,127]`, the least significant 8 bits of `x` are used after conversion to a signed integer). Note that both `NaN` and `Inf` both map to 0.

13.7.2 Example

The following piece of code demonstrates several uses of `int8`. First, the routine uses

```
--> int8(100)
ans =
  <int8>  - size: [1 1]
 100
--> int8(-100)
ans =
  <int8>  - size: [1 1]
 -100
```

In the next example, an integer outside the range of the type is passed in. The result is the 8 least significant bits of the argument.

```
--> int8(400)
ans =
  <int8>  - size: [1 1]
 -112
```

In the next example, a positive double precision argument is passed in. The result is the signed integer that is closest to the argument.

```
--> int8(pi)
ans =
  <int8>  - size: [1 1]
 3
```

In the next example, a complex argument is passed in. The result is the signed integer that is closest to the real part of the argument.

```
--> int8(5+2*i)
ans =
  <int8>  - size: [1 1]
 5
```

In the next example, a string argument is passed in. The string argument is converted into an integer array corresponding to the ASCII values of each character.

```
--> int8('helo')
ans =
  <int8>  - size: [1 4]

Columns 1 to 4
 104  101  108  111
```

In the last example, a cell-array is passed in. For cell-arrays and structure arrays, the result is an error.

```
--> int8({4})
Error: Cannot convert cell-arrays to any other type.
```

13.8 LOGICAL Convert to Logical

13.8.1 Usage

Converts the argument to a logical array. The syntax for its use is

```
   y = logical(x)
```

where `x` is an `n`-dimensional numerical array. Any nonzero element maps to a logical 1.

13.8.2 Example

Here we convert an integer array to `logical`:

```
--> logical([1,2,3,0,0,0,5,2,2])
ans =
  <logical>  - size: [1 9]
```

```
Columns 1 to 9
 1  1  1  0  0  0  1  1  1
```

The sampe example with double precision values:

```
--> logical([pi,pi,0,e,0,-1])
ans =
  <logical>  - size: [1 6]

Columns 1 to 6
 1  1  0  1  0  1
```

13.9 SINGLE Convert to 32-bit Floating Point

13.9.1 Usage

A synonym for the `float` function, converts the argument to a 32-bit floating point number. The syntax for its use is

```
   y = float(x)
```

where `x` is an `n`-dimensional numerical array. Conversion follows the general C rules. Note that both `NaN` and `Inf` are both preserved under type conversion.

13.10 STRING Convert Array to String

13.10.1 Usage

Converts the argument array into a string. The syntax for its use is

```
   y = string(x)
```

where `x` is an `n`-dimensional numerical array.

13.10.2 Example

Here we take an array containing ASCII codes for a string, and convert it into a string.

```
--> a = [104,101,108,108,111]
a =
  <int32>  - size: [1 5]

Columns 1 to 5
 104  101  108  108  111
--> string(a)
ans =
  <string>  - size: [1 5]
 hello
```

13.11 TYPEOF Determine the Type of an Argument

13.11.1 Usage

Returns a string describing the type of an array. The syntax for its use is

```
y = typeof(x),
```

The returned string is one of

- `'cell'` for cell-arrays
- `'struct'` for structure-arrays
- `'logical'` for logical arrays
- `'uint8'` for unsigned 8-bit integers
- `'int8'` for signed 8-bit integers
- `'uint16'` for unsigned 16-bit integers
- `'int16'` for signed 16-bit integers
- `'uint32'` for unsigned 32-bit integers
- `'int32'` for signed 32-bit integers
- `'float'` for 32-bit floating point numbers
- `'double'` for 64-bit floating point numbers
- `'complex'` for complex floating point numbers with 32-bits per field
- `'dcomplex'` for complex floating point numbers with 64-bits per field
- `'string'` for string arrays

13.11.2 Example

The following piece of code demonstrates the output of the `typeof` command for each possible type. The first example is with a simple cell array.

```
--> typeof({1})
ans =
  <string>  - size: [1 4]
 cell
```

The next example uses the `struct` constructor to make a simple scalar struct.

```
--> typeof(struct('foo',3))
ans =
  <string>  - size: [1 6]
 struct
```

The next example uses a comparison between two scalar integers to generate a scalar logical type.

```
--> typeof(3>5)
ans =
  <string>  - size: [1 7]
 logical
```

For the smaller integers, and the 32-bit unsigned integer types, the typecast operations are used to generate the arguments.

```
--> typeof(uint8(3))
ans =
  <string>  - size: [1 5]
 uint8
--> typeof(int8(8))
ans =
  <string>  - size: [1 4]
 int8
--> typeof(uint16(3))
ans =
  <string>  - size: [1 6]
 uint16
--> typeof(int16(8))
ans =
  <string>  - size: [1 5]
 int16
--> typeof(uint32(3))
ans =
  <string>  - size: [1 6]
 uint32
```

The 32-bit signed integer type is the default for integer arguments.

```
--> typeof(-3)
ans =
  <string>  - size: [1 5]
 int32
--> typeof(8)
ans =
  <string>  - size: [1 5]
 int32
```

Float, double, complex and double-precision complex types can be created using the suffixes.

```
--> typeof(1.0f)
ans =
  <string>  - size: [1 5]
 float
```

```
--> typeof(1.0D)
ans =
  <string>  - size: [1 6]
 double
--> typeof(1.0f+i)
ans =
  <string>  - size: [1 7]
 complex
--> typeof(1.0D+2.0D*i)
ans =
  <string>  - size: [1 8]
 dcomplex
```

13.12 UINT16 Convert to Unsigned 16-bit Integer

13.12.1 Usage

Converts the argument to an unsigned 16-bit Integer. The syntax for its use is

```
   y = uint16(x)
```

where `x` is an `n`-dimensional numerical array. Conversion follows the general C rules (e.g., if `x` is outside the normal range for an unsigned 16-bit integer of `[0,65535]`, the least significant 16 bits of `x` are used after conversion to an integer). Note that both `NaN` and `Inf` both map to 0.

13.12.2 Example

The following piece of code demonstrates several uses of `uint16`.

```
--> uint16(200)
ans =
  <uint16>  - size: [1 1]
 200
```

In the next example, an integer outside the range of the type is passed in. The result is the 16 least significant bits of the argument.

```
--> uint16(99400)
ans =
  <uint16>  - size: [1 1]
 33864
```

In the next example, a negative integer is passed in. The result is the 16 least significant bits of the argument, *after* taking the 2's complement.

```
--> uint16(-100)
ans =
  <uint16>  - size: [1 1]
 65436
```

In the next example, a positive double precision argument is passed in. The result is the unsigned integer that is closest to the argument.

```
--> uint16(pi)
ans =
  <uint16>  - size: [1 1]
 3
```

In the next example, a complex argument is passed in. The result is the unsigned integer that is closest to the real part of the argument.

```
--> uint16(5+2*i)
ans =
  <uint16>  - size: [1 1]
 5
```

In the next example, a string argument is passed in. The string argument is converted into an integer array corresponding to the ASCII values of each character.

```
--> uint16('helo')
ans =
  <uint16>  - size: [1 4]

Columns 1 to 4
 104  101  108  111
```

In the last example, a cell-array is passed in. For cell-arrays and structure arrays, the result is an error.

```
--> uint16({4})
Error: Cannot convert cell-arrays to any other type.
```

13.13 UINT32 Convert to Unsigned 32-bit Integer

13.13.1 Usage

Converts the argument to an unsigned 32-bit Integer. The syntax for its use is

```
   y = uint32(x)
```

where `x` is an `n`-dimensional numerical array. Conversion follows the general C rules (e.g., if `x` is outside the normal range for an unsigned 32-bit integer of `[0,4294967295]`, the least significant 32 bits of `x` are used after conversion to an integer). Note that both `NaN` and `Inf` both map to 0.

13.13.2 Example

The following piece of code demonstrates several uses of `uint32`.

```
--> uint32(200)
ans =
  <uint32>  - size: [1 1]
 200
```

In the next example, an integer outside the range of the type is passed in. The result is the 32 least significant bits of the argument.

```
--> uint32(40e9)
ans =
  <uint32>  - size: [1 1]
 1345294336
```

In the next example, a negative integer is passed in. The result is the 32 least significant bits of the argument, *after* taking the 2's complement.

```
--> uint32(-100)
ans =
  <uint32>  - size: [1 1]
 4294967196
```

In the next example, a positive double precision argument is passed in. The result is the unsigned integer that is closest to the argument.

```
--> uint32(pi)
ans =
  <uint32>  - size: [1 1]
 3
```

In the next example, a complex argument is passed in. The result is the unsigned integer that is closest to the real part of the argument.

```
--> uint32(5+2*i)
ans =
  <uint32>  - size: [1 1]
 5
```

In the next example, a string argument is passed in. The string argument is converted into an integer array corresponding to the ASCII values of each character.

```
--> uint32('helo')
ans =
  <uint32>  - size: [1 4]

Columns 1 to 4
 104  101  108  111
```

In the last example, a cell-array is passed in. For cell-arrays and structure arrays, the result is an error.

```
--> uint32({4})
Error: Cannot convert cell-arrays to any other type.
```

13.14 UINT8 Convert to Unsigned 8-bit Integer

13.14.1 Usage

Converts the argument to an unsigned 8-bit Integer. The syntax for its use is

```
y = uint8(x)
```

where `x` is an `n`-dimensional numerical array. Conversion follows the general C rules (e.g., if `x` is outside the normal range for an unsigned 8-bit integer of `[0,255]`, the least significant 8 bits of `x` are used after conversion to an integer). Note that both `NaN` and `Inf` both map to 0.

13.14.2 Example

The following piece of code demonstrates several uses of `uint8`.

```
--> uint8(200)
ans =
  <uint8>  - size: [1 1]
 200
```

In the next example, an integer outside the range of the type is passed in. The result is the 8 least significant bits of the argument.

```
--> uint8(400)
ans =
  <uint8>  - size: [1 1]
 144
```

In the next example, a negative integer is passed in. The result is the 8 least significant bits of the argument, *after* taking the 2's complement.

```
--> uint8(-100)
ans =
  <uint8>  - size: [1 1]
 156
```

In the next example, a positive double precision argument is passed in. The result is the unsigned integer that is closest to the argument.

```
--> uint8(pi)
ans =
  <uint8>  - size: [1 1]
 3
```

In the next example, a complex argument is passed in. The result is the unsigned integer that is closest to the real part of the argument.

```
--> uint8(5+2*i)
ans =
  <uint8>  - size: [1 1]
 5
```

In the next example, a string argument is passed in. The string argument is converted into an integer array corresponding to the ASCII values of each character.

```
--> uint8('helo')
ans =
  <uint8>  - size: [1 4]

Columns 1 to 4
 104  101  108  111
```

In the last example, a cell-array is passed in. For cell-arrays and structure arrays, the result is an error.

```
--> uint8({4})
Error: Cannot convert cell-arrays to any other type.
```

Chapter 14

Array Generation and Manipulations

14.1 BIN2DEC Convert Binary String to Decimal

14.1.1 USAGE

Converts a binary string to an integer. The syntax for its use is

```
   y = bin2dec(x)
```

where x is a binary string. If x is a matrix, then the resulting y is a column vector.

14.1.2 Example

Here we convert some numbers to bits

```
--> bin2dec('101110')
ans =
  <uint32>  - size: [1 1]
 46
--> bin2dec('010')
ans =
  <uint32>  - size: [1 1]
 2
```

14.2 BIN2INT Convert Binary Arrays to Integer

14.2.1 Usage

Converts the binary decomposition of an integer array back to an integer array. The general syntax for its use is

```
y = bin2int(x)
```

where `x` is a multi-dimensional logical array, where the last dimension indexes the bit planes (see `int2bin` for an example). By default, the output of `bin2int` is unsigned `uint32`. To get a signed integer, it must be typecast correctly.

14.2.2 Example

The following piece of code demonstrates various uses of the int2bin function. First the simplest example:

```
--> A = [2;5;6;2]
A =
  <int32>  - size: [4 1]

Columns 1 to 1
 2
 5
 6
 2
--> B = int2bin(A,8)
B =
  <logical>  - size: [4 8]

Columns 1 to 8
 0  0  0  0  0  0  1  0
 0  0  0  0  0  1  0  1
 0  0  0  0  0  1  1  0
 0  0  0  0  0  0  1  0
--> bin2int(B)
ans =
  <uint32>  - size: [4 1]

Columns 1 to 1
 2
 5
 6
 2
--> A = [1;2;-5;2]
A =
  <int32>  - size: [4 1]

Columns 1 to 1
  1
  2
 -5
  2
```

```
--> B = int2bin(A,8)
B =
  <logical>  - size: [4 8]

Columns 1 to 8
 0  0  0  0  0  0  0  1
 0  0  0  0  0  0  1  0
 1  1  1  1  1  0  1  1
 0  0  0  0  0  0  1  0
--> bin2int(B)
ans =
  <uint32>  - size: [4 1]

Columns 1 to 1
   1
   2
 251
   2
--> int32(bin2int(B))
ans =
  <int32>  - size: [4 1]

Columns 1 to 1
   1
   2
 251
   2
```

14.3 CELL Cell Array of Empty Matrices

14.3.1 Usage

Creates a cell array of empty matrix entres. Two seperate syntaxes are possible. The first syntax specifies the array dimensions as a sequence of scalar dimensions:

```
y = cell(d1,d2,...,dn).
```

The resulting array has the given dimensions, and is filled with all zeros. The type of `y` is `cell`, a cell array.

The second syntax specifies the array dimensions as a vector, where each element in the vector specifies a dimension length:

```
y = cell([d1,d2,...,dn]).
```

This syntax is more convenient for calling `zeros` using a variable for the argument. In both cases, specifying only one dimension results in a square matrix output.

14.3.2 Example

The following examples demonstrate generation of some zero arrays using the first form.

```
--> cell(2,3,2)
ans =
  <cell array> - size: [2 3 2]
(:,:,1) =

Columns 1 to 3
 []     []     []
 []     []     []
(:,:,2) =

Columns 1 to 3
 []     []     []
 []     []     []
--> cell(1,3)
ans =
  <cell array> - size: [1 3]

Columns 1 to 3
 []     []     []
```

The same expressions, using the second form.

```
--> cell([2,6])
ans =
  <cell array> - size: [2 6]

Columns 1 to 3
 []     []     []
 []     []     []

Columns 4 to 6
 []     []     []
 []     []     []
--> cell([1,3])
ans =
  <cell array> - size: [1 3]

Columns 1 to 3
 []     []     []
```

14.4 CHAR Convert to character array or string

14.4.1 Usage

The `char` function can be used to convert an array into a string. It has several forms. The first form is

```
y = char(x)
```

where `x` is a numeric array containing character codes. FreeMat does not currently support Unicode, so the character codes must be in the range of `[0,255]`. The output is a string of the same size as `x`. A second form is

```
y = char(c)
```

where `c` is a cell array of strings, creates a matrix string where each row contains a string from the corresponding cell array. The third form is

```
y = char(s1, s2, s3, ...)
```

where `si` are a character arrays. The result is a matrix string where each row contains a string from the corresponding argument.

14.4.2 Example

Here is an example of the first technique being used to generate a string containing some ASCII characters

```
--> char([32:64;65:97])
ans =
  <string>  - size: [2 33]
  !"#$%&'()*+,-./0123456789:;<=>?@
 ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`a
```

In the next example, we form a character array from a set of strings in a cell array. Note that the character array is padded with spaces to make the rows all have the same length.

```
--> char({'hello','to','the','world'})
ans =
  <string>  - size: [4 5]
 hello
 to
 the
 world
```

In the last example, we pass the individual strings as explicit arguments to `char`

```
--> char('hello','to','the','world')
ans =
  <string>  - size: [4 5]
 hello
 to
 the
 world
```

14.5 CIRCSHIFT Circularly Shift an Array

14.5.1 USAGE

Applies a circular shift along each dimension of a given array. The syntax for its use is

```
  y = circshift(x,shiftvec)
```

where `x` is an n-dimensional array, and `shiftvec` is a vector of integers, each of which specify how much to shift `x` along the corresponding dimension.

14.5.2 Example

The following examples show some uses of `circshift` on N-dimensional arrays.

```
--> x = int32(rand(4,5)*10)
x =
  <int32>  - size: [4 5]

Columns 1 to 5
 6  9  9  3  1
 3  5  0  4  7
 5  0  2  6  5
 1  4  0  4  0
--> circshift(x,[1,0])
ans =
  <int32>  - size: [4 5]

Columns 1 to 5
 1  4  0  4  0
 6  9  9  3  1
 3  5  0  4  7
 5  0  2  6  5
--> circshift(x,[0,-1])
ans =
  <int32>  - size: [4 5]

Columns 1 to 5
 9  9  3  1  6
 5  0  4  7  3
 0  2  6  5  5
 4  0  4  0  1
--> circshift(x,[2,2])
ans =
  <int32>  - size: [4 5]

Columns 1 to 5
```

```
 6  5  5  0  2
 4  0  1  4  0
 3  1  6  9  9
 4  7  3  5  0
--> x = int32(rand(4,5,3)*10)
x =
  <int32>  - size: [4 5 3]
(:,:,1) =

Columns 1 to 5
 5  6  9  1  5
 9  6  4  6  2
 2  0  6  8  7
 8  5  4  3  0
(:,:,2) =

Columns 1 to 5
 1  4  4  9  3
 3  5  2  7  1
 1  6  7  6  1
 9  5  5  5  2
(:,:,3) =

Columns 1 to 5
 1  1  0  9  0
 5  7  8  2  5
 3  8  3  3  6
 9  4  7  4  2
--> circshift(x,[1,0,0])
ans =
  <int32>  - size: [4 5 3]
(:,:,1) =

Columns 1 to 5
 8  5  4  3  0
 5  6  9  1  5
 9  6  4  6  2
 2  0  6  8  7
(:,:,2) =

Columns 1 to 5
 9  5  5  5  2
 1  4  4  9  3
 3  5  2  7  1
 1  6  7  6  1
(:,:,3) =
```

```
Columns 1 to 5
 9  4  7  4  2
 1  1  0  9  0
 5  7  8  2  5
 3  8  3  3  6
--> circshift(x,[0,-1,0])
ans =
  <int32>  - size: [4 5 3]
(:,:,1) =

Columns 1 to 5
 6  9  1  5  5
 6  4  6  2  9
 0  6  8  7  2
 5  4  3  0  8
(:,:,2) =

Columns 1 to 5
 4  4  9  3  1
 5  2  7  1  3
 6  7  6  1  1
 5  5  5  2  9
(:,:,3) =

Columns 1 to 5
 1  0  9  0  1
 7  8  2  5  5
 8  3  3  6  3
 4  7  4  2  9
--> circshift(x,[0,0,-1])
ans =
  <int32>  - size: [4 5 3]
(:,:,1) =

Columns 1 to 5
 1  4  4  9  3
 3  5  2  7  1
 1  6  7  6  1
 9  5  5  5  2
(:,:,2) =

Columns 1 to 5
 1  1  0  9  0
 5  7  8  2  5
 3  8  3  3  6
```

```
 9  4  7  4  2
(:,:,3) =

Columns 1 to 5
 5  6  9  1  5
 9  6  4  6  2
 2  0  6  8  7
 8  5  4  3  0
--> circshift(x,[2,-3,1])
ans =
  <int32>  - size: [4 5 3]
(:,:,1) =

Columns 1 to 5
 3  6  3  8  3
 4  2  9  4  7
 9  0  1  1  0
 2  5  5  7  8
(:,:,2) =

Columns 1 to 5
 8  7  2  0  6
 3  0  8  5  4
 1  5  5  6  9
 6  2  9  6  4
(:,:,3) =

Columns 1 to 5
 6  1  1  6  7
 5  2  9  5  5
 9  3  1  4  4
 7  1  3  5  2
```

14.6 COND Condition Number of a Matrix

14.6.1 Usage

Calculates the condition number of a matrix. To compute the 2-norm condition number of a matrix (ratio of largest to smallest singular values), use the syntax

```
y = cond(A)
```

where A is a matrix. If you want to compute the condition number in a different norm (e.g., the 1-norm), use the second syntax

```
y = cond(A,p)
```

where `p` is the norm to use when computing the condition number. The following choices of `p` are supported

- `p = 1` returns the 1-norm, or the max column sum of A
- `p = 2` returns the 2-norm (largest singular value of A)
- `p = inf` returns the infinity norm, or the max row sum of A
- `p = 'fro'` returns the Frobenius-norm (vector Euclidean norm, or RMS value)

14.6.2 Function Internals

The condition number is defined as

$$\frac{\|A\|_p}{\|A^{-1}\|_p}$$

This equation is precisely how the condition number is computed for the case `p ~= 2`. For the `p=2` case, the condition number can be computed much more efficiently using the ratio of the largest and smallest singular values.

14.6.3 Example

The condition number of this matrix is large

```
--> A = [1,1;0,1e-15]
A =
  <double>  - size: [2 2]

Columns 1 to 2
 1.000000000000000  1.000000000000000
 0.000000000000000  0.000000000000001
--> cond(A)
ans =
  <double>  - size: [1 1]
 2000000000000000
--> cond(A,1)
ans =
  <double>  - size: [1 1]
 2000000000000002
```

You can also (for the case `p=1` use `rcond` to calculate an estimate of the condition number

```
--> 1/rcond(A)
ans =
  <double>  - size: [1 1]
 2000000000000001.8
```

14.7 DEC2BIN Convert Decimal to Binary String

14.7.1 USAGE

Converts an integer to a binary string. The syntax for its use is

```
y = dec2bin(x,n)
```

where `x` is the positive integer, and `n` is the number of bits to use in the representation. Alternately, if you leave `n` unspecified,

```
y = dec2bin(x)
```

the minimum number of bits needed to represent `x` are used. If `x` is a vector, then the resulting `y` is a character matrix.

14.7.2 Example

Here we convert some numbers to bits

```
--> dec2bin(56)
ans =
  <string>  - size: [1 6]
 111000
--> dec2bin(1039456)
ans =
  <string>  - size: [1 20]
 11111101110001100000
--> dec2bin([63,73,32],5)
ans =
  <string>  - size: [3 5]
 11111
 01001
 00000
```

14.8 DET Determinant of a Matrix

14.8.1 Usage

Calculates the determinant of a matrix. Note that for all but very small problems, the determinant is not particularly useful. The condition number `cond` gives a more reasonable estimate as to the suitability of a matrix for inversion than comparing `det(A)` to zero. In any case, the syntax for its use is

```
y = det(A)
```

where A is a square matrix.

14.8.2 Function Internals

The determinant is calculated via the `LU` decomposition. Note that the determinant of a product of matrices is the product of the determinants. Then, we have that

$$LU = PA$$

where `L` is lower triangular with 1s on the main diagonal, `U` is upper triangular, and `P` is a row-permutation matrix. Taking the determinant of both sides yields

$$|LU| = |L||U| = |U| = |PA| = |P||A|$$

where we have used the fact that the determinant of `L` is 1. The determinant of `P` (which is a row exchange matrix) is either 1 or -1.

14.8.3 Example

Here we assemble a random matrix and compute its determinant

```
--> A = rand(5);
--> det(A)
ans =
  <double>  - size: [1 1]
 -0.05210265560352309
```

Then, we exchange two rows of `A` to demonstrate how the determinant changes sign (but the magnitude is the same)

```
--> B = A([2,1,3,4,5],:);
--> det(B)
ans =
  <double>  - size: [1 1]
 0.05210265560352309
```

14.9 DIAG Diagonal Matrix Construction/Extraction

14.9.1 Usage

The `diag` function is used to either construct a diagonal matrix from a vector, or return the diagonal elements of a matrix as a vector. The general syntax for its use is

```
  y = diag(x,n)
```

If `x` is a matrix, then `y` returns the `n`-th diagonal. If `n` is omitted, it is assumed to be zero. Conversely, if `x` is a vector, then `y` is a matrix with `x` set to the `n`-th diagonal.

14.9.2 Examples

Here is an example of `diag` being used to extract a diagonal from a matrix.

```
--> A = int32(10*rand(4,5))
A =
  <int32>  - size: [4 5]

Columns 1 to 5
 7  3  9  9  0
 4  4  3  9  5
 2  4  9  2  7
 2  6  1  9  8
--> diag(A)
ans =
  <int32>  - size: [4 1]

Columns 1 to 1
 7
 4
 9
 9
--> diag(A,1)
ans =
  <int32>  - size: [4 1]

Columns 1 to 1
 3
 3
 2
 8
```

Here is an example of the second form of `diag`, being used to construct a diagonal matrix.

```
--> x = int32(10*rand(1,3))
x =
  <int32>  - size: [1 3]

Columns 1 to 3
 8  7  9
--> diag(x)
ans =
  <int32>  - size: [3 3]

Columns 1 to 3
 8  0  0
 0  7  0
```

```
 0  0  9
--> diag(x,-1)
ans =
  <int32>  - size: [4 4]

Columns 1 to 4
 0  0  0  0
 8  0  0  0
 0  7  0  0
 0  0  9  0
```

14.10 EYE Identity Matrix

14.10.1 USAGE

Creates an identity matrix of the specified size. The syntax for its use is

```
y = eye(n)
```

where `n` is the size of the identity matrix. The type of the output matrix is `float`.

14.10.2 Example

The following example demonstrates the identity matrix.

```
--> eye(3)
ans =
  <float>  - size: [3 3]

Columns 1 to 3
 1  0  0
 0  1  0
 0  0  1
```

14.11 FIND Find Non-zero Elements of An Array

14.11.1 Usage

Returns a vector that contains the indicies of all non-zero elements in an array. The usage is

```
y = find(x)
```

The indices returned are generalized column indices, meaning that if the array `x` is of size `[d1,d2,...,dn]`, and the element `x(i1,i2,...,in)` is nonzero, then `y` will contain the integer

$$i_1 + (i_2 - 1)d_1 + (i_3 - 1)d_1 d_2 + \dots$$

The second syntax for the `find` command is

```
[r,c] = find(x)
```

which returns the row and column index of the nonzero entries of `x`. The third syntax for the `find` command also returns the values

```
[r,c,v] = find(x).
```

This form is particularly useful for converting sparse matrices into IJV form.

14.11.2 Example

Some simple examples of its usage, and some common uses of `find` in FreeMat programs.

```
--> a = [1,2,5,2,4];
--> find(a==2)
ans =
  <uint32>  - size: [2 1]

Columns 1 to 1
 2
 4
```

Here is an example of using find to replace elements of `A` that are `0` with the number `5`.

```
--> A = [1,0,3;0,2,1;3,0,0]
A =
  <int32>  - size: [3 3]

Columns 1 to 3
 1  0  3
 0  2  1
 3  0  0
--> n = find(A==0)
n =
  <uint32>  - size: [4 1]

Columns 1 to 1
 2
 4
 6
 9
--> A(n) = 5
A =
  <int32>  - size: [3 3]

Columns 1 to 3
 1  5  3
 5  2  1
 3  5  5
```

Incidentally, a better way to achieve the same concept is:

```
--> A = [1,0,3;0,2,1;3,0,0]
A =
  <int32>  - size: [3 3]

Columns 1 to 3
 1  0  3
 0  2  1
 3  0  0
--> A(A==0) = 5
A =
  <int32>  - size: [3 3]

Columns 1 to 3
 1  5  3
 5  2  1
 3  5  5
```

Now, we can also return the indices as row and column indices using the two argument form of `find`:

```
--> A = [1,0,3;0,2,1;3,0,0]
A =
  <int32>  - size: [3 3]

Columns 1 to 3
 1  0  3
 0  2  1
 3  0  0
--> [r,c] = find(A)
r =
  <uint32>  - size: [5 1]

Columns 1 to 1
 1
 3
 2
 1
 2
c =
  <uint32>  - size: [5 1]

Columns 1 to 1
 1
 1
 2
 3
```

```
 3
```

Or the three argument form of `find`, which returns the value also:

```
--> [r,c,v] = find(A)
r =
  <uint32>  - size: [5 1]

Columns 1 to 1
 1
 3
 2
 1
 2
c =
  <uint32>  - size: [5 1]

Columns 1 to 1
 1
 1
 2
 3
 3
v =
  <int32>  - size: [5 1]

Columns 1 to 1
 1
 3
 2
 3
 1
```

14.12 FLIPDIM Reverse a Matrix Along a Given Dimension

14.12.1 USAGE

Reverses an array along the given dimension. The syntax for its use is

```
   y = flipdim(x,n)
```

where `x` is matrix, and `n` is the dimension to reverse.

14.12.2 Example

The following examples show some uses of `flipdim` on N-dimensional arrays.

```
--> x = int32(rand(4,5,3)*10)
x =
  <int32>  - size: [4 5 3]
(:,:,1) =

Columns 1 to 5
 1  2  5  7  9
 9  6  5  1  4
 3  9  3  4  3
 5  2  2  1  2
(:,:,2) =

Columns 1 to 5
 9  7  5  3  0
 1  7  9  4  7
 3  5  4  9  4
 5  5  8  8  6
(:,:,3) =

Columns 1 to 5
 7  5  4  0  0
 4  8  7  8  1
 0  9  8  9  4
 4  7  3  4  8
--> flipdim(x,1)
ans =
  <int32>  - size: [4 5 3]
(:,:,1) =

Columns 1 to 5
 5  2  2  1  2
 3  9  3  4  3
 9  6  5  1  4
 1  2  5  7  9
(:,:,2) =

Columns 1 to 5
 5  5  8  8  6
 3  5  4  9  4
 1  7  9  4  7
 9  7  5  3  0
(:,:,3) =

Columns 1 to 5
 4  7  3  4  8
 0  9  8  9  4
```

```
 4  8  7  8  1
 7  5  4  0  0
--> flipdim(x,2)
ans =
  <int32>  - size: [4 5 3]
(:,:,1) =

Columns 1 to 5
 9  7  5  2  1
 4  1  5  6  9
 3  4  3  9  3
 2  1  2  2  5
(:,:,2) =

Columns 1 to 5
 0  3  5  7  9
 7  4  9  7  1
 4  9  4  5  3
 6  8  8  5  5
(:,:,3) =

Columns 1 to 5
 0  0  4  5  7
 1  8  7  8  4
 4  9  8  9  0
 8  4  3  7  4
--> flipdim(x,3)
ans =
  <int32>  - size: [4 5 3]
(:,:,1) =

Columns 1 to 5
 7  5  4  0  0
 4  8  7  8  1
 0  9  8  9  4
 4  7  3  4  8
(:,:,2) =

Columns 1 to 5
 9  7  5  3  0
 1  7  9  4  7
 3  5  4  9  4
 5  5  8  8  6
(:,:,3) =

Columns 1 to 5
```

```
 1  2  5  7  9
 9  6  5  1  4
 3  9  3  4  3
 5  2  2  1  2
```

14.13 FLIPLR Reverse the Columns of a Matrix

14.13.1 USAGE

Reverses the columns of a matrix. The syntax for its use is

```
y = fliplr(x)
```

where `x` is matrix. If `x` is an N-dimensional array then the second dimension is reversed.

14.13.2 Example

The following example shows `fliplr` applied to a 2D matrix.

```
--> x = int32(rand(4)*10)
x =
  <int32>  - size: [4 4]

Columns 1 to 4
 4  4  2  5
 9  4  8  4
 7  2  0  2
 0  7  0  2
--> fliplr(x)
ans =
  <int32>  - size: [4 4]

Columns 1 to 4
 5  2  4  4
 4  8  4  9
 2  0  2  7
 2  0  7  0
```

For a 3D array, note how the columns in each slice are flipped.

```
--> x = int32(rand(4,4,3)*10)
x =
  <int32>  - size: [4 4 3]
(:,:,1) =

Columns 1 to 4
 5  9  7  5
```

```
 2  3  6  4
 8  7  2  8
 3  1  5  4
(:,:,2) =

Columns 1 to 4
 7  9  2  8
 3  7  9  4
 3  3  2  6
 0  1  9  4
(:,:,3) =

Columns 1 to 4
 8  7  3  0
 8  6  2  1
 7  0  8  1
 4  2  6  3
--> fliplr(x)
ans =
  <int32>  - size: [4 4 3]
(:,:,1) =

Columns 1 to 4
 5  7  9  5
 4  6  3  2
 8  2  7  8
 4  5  1  3
(:,:,2) =

Columns 1 to 4
 8  2  9  7
 4  9  7  3
 6  2  3  3
 4  9  1  0
(:,:,3) =

Columns 1 to 4
 0  3  7  8
 1  2  6  8
 1  8  0  7
 3  6  2  4
```

14.14 FLIPUD Reverse the Columns of a Matrix

14.14.1 USAGE

Reverses the rows of a matrix. The syntax for its use is

```
y = flipud(x)
```

where `x` is matrix. If `x` is an N-dimensional array then the first dimension is reversed.

14.14.2 Example

The following example shows `flipud` applied to a 2D matrix.

```
--> x = int32(rand(4)*10)
x =
  <int32>  - size: [4 4]

Columns 1 to 4
 9  9  6  3
 8  0  7  3
 1  9  8  5
 3  2  3  3
--> flipud(x)
ans =
  <int32>  - size: [4 4]

Columns 1 to 4
 3  2  3  3
 1  9  8  5
 8  0  7  3
 9  9  6  3
```

For a 3D array, note how the rows in each slice are flipped.

```
--> x = int32(rand(4,4,3)*10)
x =
  <int32>  - size: [4 4 3]
(:,:,1) =

Columns 1 to 4
 3  7  1  7
 8  9  7  4
 5  0  5  4
 3  1  9  6
(:,:,2) =

Columns 1 to 4
```

```
 8  5  5  4
 9  6  8  9
 2  8  2  4
 7  5  3  8
(:,:,3) =

Columns 1 to 4
 9  7  0  0
 4  9  7  1
 0  6  2  4
 3  3  0  1
--> flipud(x)
ans =
  <int32>  - size: [4 4 3]
(:,:,1) =

Columns 1 to 4
 3  1  9  6
 5  0  5  4
 8  9  7  4
 3  7  1  7
(:,:,2) =

Columns 1 to 4
 7  5  3  8
 2  8  2  4
 9  6  8  9
 8  5  5  4
(:,:,3) =

Columns 1 to 4
 3  3  0  1
 0  6  2  4
 4  9  7  1
 9  7  0  0
```

14.15 INT2BIN Convert Integer Arrays to Binary

14.15.1 Usage

Computes the binary decomposition of an integer array to the specified number of bits. The general syntax for its use is

```
y = int2bin(x,n)
```

where `x` is a multi-dimensional integer array, and `n` is the number of bits to expand it to. The output array `y` has one extra dimension to it than the input. The bits are expanded along this extra dimension.

14.15.2 Example

The following piece of code demonstrates various uses of the int2bin function. First the simplest example:

```
--> A = [2;5;6;2]
A =
  <int32>  - size: [4 1]

Columns 1 to 1
 2
 5
 6
 2
--> int2bin(A,8)
ans =
  <logical>  - size: [4 8]

Columns 1 to 8
 0  0  0  0  0  0  1  0
 0  0  0  0  0  1  0  1
 0  0  0  0  0  1  1  0
 0  0  0  0  0  0  1  0
--> A = [1;2;-5;2]
A =
  <int32>  - size: [4 1]

Columns 1 to 1
  1
  2
 -5
  2
--> int2bin(A,8)
ans =
  <logical>  - size: [4 8]

Columns 1 to 8
 0  0  0  0  0  0  0  1
 0  0  0  0  0  0  1  0
 1  1  1  1  1  0  1  1
 0  0  0  0  0  0  1  0
```

14.16 ISCELL Test For Cell Array

14.16.1 Usage

The syntax for `iscell` is

```
x = iscell(y)
```

and it returns a logical 1 if the argument is a cell array and a logical 0 otherwise.

14.16.2 Example

Here are some examples of `iscell`

```
--> iscell('foo')
ans =
  <logical>  - size: [1 1]
 0
--> iscell(2)
ans =
  <logical>  - size: [1 1]
 0
--> iscell({1,2,3})
ans =
  <logical>  - size: [1 1]
 1
```

14.17 ISCELLSTR Test For Cell Array of Strings

14.17.1 Usage

The syntax for `iscellstr` is

```
x = iscellstr(y)
```

and it returns a logical 1 if the argument is a cell array in which every cell is a character array (or is empty), and a logical 0 otherwise.

14.17.2 Example

Here is a simple example

```
--> A = {'Hello','Yellow';'Mellow','Othello'}
A =
  <cell array> - size: [2 2]

Columns 1 to 2
 Hello    Yellow
```

```
 Mellow    Othello
--> iscellstr(A)
ans =
  <logical>  - size: [1 1]
 1
```

14.18 ISCHAR Test For Character Array (string)

14.18.1 Usage

The syntax for `ischar` is

```
   x = ischar(y)
```

and it returns a logical 1 if the argument is a string and a logical 0 otherwise.

14.19 ISLOGICAL Test for Logical Array

14.19.1 Usage

The syntax for `islogical` is

```
   x = islogical(y)
```

and it returns a logical 1 if the argument is a logical array and a logical 0 otherwise.

14.20 ISNUMERIC Test for Numeric Array

14.20.1 Usage

The syntax for `isnumeric` is

```
  x = isnumeric(y)
```

and it returns a logical 1 if the argument is a numeric (i.e., not a structure array, cell array, string or user defined class), and a logical 0 otherwise.

14.21 ISREAL Test For Real Array

14.21.1 Usage

The syntax for `isreal` is

```
   x = isreal(y)
```

and it returns a logical 1 if the argument is a real type (integer, float, or double), and a logical 0 otherwise.

14.22 ISSCALAR Test For Scalar

14.22.1 Usage

The syntax for `isscalar` is

```
x = isscalar(y)
```

and it returns a logical 1 if the argument is a scalar, and a logical 0 otherwise.

14.23 ISSTR Test For Character Array (string)

14.23.1 Usage

The syntax for `isstr` is

```
x = isstr(y)
```

and it returns a logical 1 if the argument is a string and a logical 0 otherwise.

14.24 ISSTRUCT Test For Structure Array

14.24.1 Usage

The syntax for `isstruct` is

```
x = isstruct(y)
```

and it returns a logical 1 if the argument is a structure array, and a logical 0 otherwise.

14.25 LENGTH Length of an Array

14.25.1 Usage

Returns the length of an array `x`. The syntax for its use is

```
y = length(x)
```

and is defined as the maximum length of `x` along any of its dimensions, i.e., `max(size(x))`. If you want to determine the number of elements in `x`, use the `numel` function instead.

14.25.2 Example

For a `4 x 4 x 3` matrix, the length is `4`, not `48`, as you might expect.

```
--> x = rand(4,4,3);
--> length(x)
ans =
  <uint32>  - size: [1 1]
 4
```

14.26 LINSPACE Linearly Spaced Vector

14.26.1 Usage

Generates a row vector with the specified number of elements, with entries uniformly spaced between two specified endpoints. The syntax for its use is either

```
y = linspace(a,b,count)
```

or, for a default `count = 100`,

```
y = linspace(a,b);
```

14.26.2 Examples

Here is a simple example of using `linspace`

```
--> x = linspace(0,1,5)
x =
  <double>  - size: [1 5]

Columns 1 to 5
 0.00  0.25  0.50  0.75  1.00
```

14.27 NDGRID Generate N-Dimensional Grid

14.27.1 Usage

Generates N-dimensional grids, each of which is constant in all but one dimension. The syntax for its use is either

```
[y1, y2, ..., ym] = ndgrid(x1, x2, ..., xn)
```

where `m <= n` or

```
[y1, y2, ..., ym] = ndgrid(x1)
```

which is equivalent to the first form, with `x1=x2=...=xn`. Each output `yi` is an `n`-dimensional array, with values such that

$$y_i(d_1, \ldots, d_{i-1}, d_i, d_{i+1}, \ldots, d_m) = x_i(d_i)$$

`ndgrid` is useful for evaluating multivariate functionals over a range of arguments. It is a generalization of `meshgrid`, except that `meshgrid` transposes the dimensions corresponding to the first two arguments to better fit graphical applications.

14.27.2 Example

Here is a simple `ndgrid` example

```
--> [a,b] = ndgrid(1:2,3:5)
a =
  <int32>  - size: [2 3]

Columns 1 to 3
 1  1  1
 2  2  2
b =
  <int32>  - size: [2 3]

Columns 1 to 3
 3  4  5
 3  4  5
--> [a,b,c] = ndgrid(1:2,3:5,0:1)
a =
  <int32>  - size: [2 3 2]
(:,:,1) =

Columns 1 to 3
 1  1  1
 2  2  2
(:,:,2) =

Columns 1 to 3
 1  1  1
 2  2  2
b =
  <int32>  - size: [2 3 2]
(:,:,1) =

Columns 1 to 3
 3  4  5
 3  4  5
(:,:,2) =

Columns 1 to 3
 3  4  5
 3  4  5
c =
  <int32>  - size: [2 3 2]
(:,:,1) =

Columns 1 to 3
```

```
 0  0  0
 0  0  0
(:,:,2) =

Columns 1 to 3
 1  1  1
 1  1  1
```

Here we use the second form

```
--> [a,b,c] = ndgrid(1:3)
a =
  <int32>  - size: [3 3 3]
(:,:,1) =

Columns 1 to 3
 1  1  1
 2  2  2
 3  3  3
(:,:,2) =

Columns 1 to 3
 1  1  1
 2  2  2
 3  3  3
(:,:,3) =

Columns 1 to 3
 1  1  1
 2  2  2
 3  3  3
b =
  <int32>  - size: [3 3 3]
(:,:,1) =

Columns 1 to 3
 1  2  3
 1  2  3
 1  2  3
(:,:,2) =

Columns 1 to 3
 1  2  3
 1  2  3
 1  2  3
(:,:,3) =
```

```
Columns 1 to 3
 1  2  3
 1  2  3
 1  2  3
c =
  <int32>  - size: [3 3 3]
(:,:,1) =

Columns 1 to 3
 1  1  1
 1  1  1
 1  1  1
(:,:,2) =

Columns 1 to 3
 2  2  2
 2  2  2
 2  2  2
(:,:,3) =

Columns 1 to 3
 3  3  3
 3  3  3
 3  3  3
```

14.28 NDIMS Number of Dimensions in Array

14.28.1 Usage

The `ndims` function returns the number of dimensions allocated in an array. The general syntax for its use is

```
  n = ndims(x)
```

and is equivalent to `length(size(x))`.

14.29 NONZEROS Retrieve Nonzero Matrix Entries

14.29.1 USAGE

Returns a dense column vector containing the nonzero elements of the argument matrix. The syntax for its use is

```
   y = nonzeros(x)
```

where `x` is the argument array. The argument matrix may be sparse as well as dense.

14.29.2 Example

Here is an example of using `nonzeros` on a sparse matrix.

```
--> a = rand(8); a(a>0.2) = 0;
--> A = sparse(a)
A =
  <double>  - size: [8 8]
Matrix is sparse with 8 nonzeros
--> nonzeros(A)
ans =
  <double>  - size: [8 1]

Columns 1 to 1
 0.098953370131552809
 0.066184717220581502
 0.021949046494394442
 0.113011009385646233
 0.144412295716459083
 0.107450039193461300
 0.008047519987897545
 0.076519530544516101
```

14.30 NORM Norm Calculation

14.30.1 Usage

Calculates the norm of a matrix. There are two ways to use the `norm` function. The general syntax is

```
y = norm(A,p)
```

where `A` is the matrix to analyze, and `p` is the type norm to compute. The following choices of `p` are supported

- `p = 1` returns the 1-norm, or the max column sum of A
- `p = 2` returns the 2-norm (largest singular value of A)
- `p = inf` returns the infinity norm, or the max row sum of A
- `p = 'fro'` returns the Frobenius-norm (vector Euclidean norm, or RMS value)

For a vector, the regular norm calculations are performed:

- `1 <= p < inf` returns `sum(abs(A).^p)^(1/p)`
- `p` unspecified returns `norm(A,2)`
- `p = inf` returns max(abs(A))
- `p = -inf` returns min(abs(A))

14.30.2 Examples

Here are the various norms calculated for a sample matrix

```
--> A = float(rand(3,4))
A =
  <float>  - size: [3 4]

Columns 1 to 4
 0.113413215  0.778945625  0.951759875  0.861983240
 0.206183329  0.367634743  0.910258651  0.094072342
 0.233640403  0.972906291  0.578187644  0.817632318
--> norm(A,1)
ans =
  <float>  - size: [1 1]
 2.440206
--> norm(A,2)
ans =
  <float>  - size: [1 1]
 2.203075
--> norm(A,inf)
ans =
  <float>  - size: [1 1]
 2.706102
--> norm(A,'fro')
ans =
  <float>  - size: [1 1]
 2.299462
```

Next, we calculate some vector norms.

```
--> A = float(rand(4,1))
A =
  <float>  - size: [4 1]

Columns 1 to 1
 0.35852790
 0.61784583
 0.63974983
 0.79405016
--> norm(A,1)
ans =
  <double>  - size: [1 1]
 2.4101736545562744
--> norm(A,2)
ans =
  <float>  - size: [1 1]
```

```
 1.2450186
--> norm(A,7)
ans =
  <double>  - size: [1 1]
 0.8328831813550448
--> norm(A,inf)
ans =
  <float>  - size: [1 1]
 0.79405016
--> norm(A,-inf)
ans =
  <float>  - size: [1 1]
 0.3585279
```

14.31 NUMEL Number of Elements in an Array

14.31.1 Usage

Returns the number of elements in an array `x`, or in a subindex expression. The syntax for its use is either

```
   y = numel(x)
```

or

```
   y = numel(x,varargin)
```

Generally, `numel` returns `prod(size(x))`, the number of total elements in `x`. However, you can specify a number of indexing expressions for `varagin` such as `index1, index2, ..., indexm`. In that case, the output of `numel` is `prod(size(x(index1,...,indexm)))`.

14.31.2 Example

For a `4 x 4 x 3` matrix, the length is `4`, not `48`, as you might expect, but `numel` is `48`.

```
--> x = rand(4,4,3);
--> length(x)
ans =
  <uint32>  - size: [1 1]
 4
--> numel(x)
ans =
  <int32>  - size: [1 1]
 48
```

Here is an example of using `numel` with indexing expressions.

```
--> numel(x,1:3,1:2,2)
ans =
  <int32>  - size: [1 1]
 6
```

14.32 ONES Array of Ones

14.32.1 Usage

Creates an array of ones of the specified size. Two seperate syntaxes are possible. The first syntax specifies the array dimensions as a sequence of scalar dimensions:

```
y = ones(d1,d2,...,dn).
```

The resulting array has the given dimensions, and is filled with all ones. The type of `y` is `float`, a 32-bit floating point array. To get arrays of other types, use the typecast functions (e.g., `uint8`, `int8`, etc.).

The second syntax specifies the array dimensions as a vector, where each element in the vector specifies a dimension length:

```
y = ones([d1,d2,...,dn]).
```

This syntax is more convenient for calling `ones` using a variable for the argument. In both cases, specifying only one dimension results in a square matrix output.

14.32.2 Example

The following examples demonstrate generation of some arrays of ones using the first form.

```
--> ones(2,3,2)
ans =
  <float>  - size: [2 3 2]
(:,:,1) =

Columns 1 to 3
 1  1  1
 1  1  1
(:,:,2) =

Columns 1 to 3
 1  1  1
 1  1  1
--> ones(1,3)
ans =
  <float>  - size: [1 3]

Columns 1 to 3
 1  1  1
```

The same expressions, using the second form.

```
--> ones([2,6])
ans =
  <float>  - size: [2 6]

Columns 1 to 6
 1  1  1  1  1  1
 1  1  1  1  1  1
--> ones([1,3])
ans =
  <float>  - size: [1 3]

Columns 1 to 3
 1  1  1
```

Finally, an example of using the type casting function `uint16` to generate an array of 16-bit unsigned integers with a value of 1.

```
--> uint16(ones(3))
ans =
  <uint16>  - size: [3 3]

Columns 1 to 3
 1  1  1
 1  1  1
 1  1  1
```

14.33 PINV Moore-Penrose Pseudoinverse

14.33.1 Usage

Calculates the Moore-Penrose pseudoinverse of a matrix. The general syntax for its use is

```
y = pinv(A,tol)
```

or for a default specification of the tolerance `tol`,

```
y = pinv(A)
```

For any `m x n` matrix `A`, the Moore-Penrose pseudoinverse is the unique `n x m` matrix `B` that satisfies the following four conditions

- `A B A = A`
- `B A B = B`
- `(A B)' = A B`
- `(B A)' = B A`

Also, it is true that `B y` is the minimum norm, least squares solution to `A x = y`. The Moore-Penrose pseudoinverse is computed from the singular value decomposition of `A`, with singular values smaller than `tol` being treated as zeros. If `tol` is not specified then it is chosen as

```
tol = max(size(A)) * norm(A) * teps(A).
```

14.33.2 Function Internals

The calculation of the MP pseudo-inverse is almost trivial once the svd of the matrix is available. First, for a real, diagonal matrix with positive entries, the pseudo-inverse is simply

$$\left(\Sigma^{+}\right)_{ii} = \begin{cases} 1/\sigma_{ii} & \sigma_{ii} > 0 \\ 0 & \text{else} \end{cases}$$

One can quickly verify that this choice of matrix satisfies the four properties of the pseudoinverse. Then, the pseudoinverse of a general matrix `A = U S V'` is defined as

$$A^{+} = VS^{+}U'$$

and again, using the facts that `U' U = I` and `V V' = I`, one can quickly verify that this choice of pseudoinverse satisfies the four defining properties of the MP pseudoinverse. Note that in practice, the diagonal pseudoinverse `S^{+}` is computed with a threshold (the `tol` argument to `pinv`) so that singular values smaller than `tol` are treated like zeros.

14.33.3 Examples

Consider a simple `1 x 2` matrix example, and note the various Moore-Penrose conditions:

```
--> A = float(rand(1,2))
A =
  <float>  - size: [1 2]

Columns 1 to 2
 0.93414986  0.88431287
--> B = pinv(A)
B =
  <float>  - size: [2 1]

Columns 1 to 1
 0.56456208
 0.53444266
--> A*B*A
ans =
  <float>  - size: [1 2]

Columns 1 to 2
 0.934150  0.884313
```

```
--> B*A*B
ans =
  <float>  - size: [2 1]

Columns 1 to 1
 0.56456214
 0.53444272
--> A*B
ans =
  <float>  - size: [1 1]
 1.0000001
--> B*A
ans =
  <float>  - size: [2 2]

Columns 1 to 2
 0.52738559  0.49924952
 0.49924955  0.47261453
```

To demonstrate that `pinv` returns the least squares solution, consider the following very simple case

```
--> A = float([1;1;1;1])
A =
  <float>  - size: [4 1]

Columns 1 to 1
 1
 1
 1
 1
```

The least squares solution to `A x = b` is just `x = mean(b)`, and computing the `pinv` of `A` demonstrates this

```
--> pinv(A)
ans =
  <float>  - size: [1 4]

Columns 1 to 4
 0.25  0.25  0.25  0.25
```

Similarly, we can demonstrate the minimum norm solution with the following simple case

```
--> A = float([1,1])
A =
  <float>  - size: [1 2]

Columns 1 to 2
 1  1
```

The solutions of `A x = 5` are those `x_1` and `x_2` such that `x_1 + x_2 = 5`. The norm of `x` is `x_1^ + x_2^2`, which is `x_1^2 + (5-x_1)^2`, which is minimized for `x_1 = x_2 = 2.5`:

```
--> pinv(A) * 5.0f
ans =
  <float>  - size: [2 1]

Columns 1 to 1
 2.5
 2.5
```

14.34 RANK Calculate the Rank of a Matrix

14.34.1 Usage

Returns the rank of a matrix. There are two ways to use the `rank` function is

```
   y = rank(A,tol)
```

where `tol` is the tolerance to use when computing the rank. The second form is

```
   y = rank(A)
```

in which case the tolerance `tol` is chosen as

```
   tol = max(size(A))*max(s)*eps,
```

where `s` is the vector of singular values of `A`. The rank is computed using the singular value decomposition `svd`.

14.34.2 Examples

Some examples of matrix rank calculations

```
--> rank([1,3,2;4,5,6])
ans =
  <int32>  - size: [1 1]
 2
--> rank([1,2,3;2,4,6])
ans =
  <int32>  - size: [1 1]
 1
```

Here we construct an ill-conditioned matrix, and show the use of the `tol` argument.

```
--> A = [1,0;0,eps/2]
A =
  <double>  - size: [2 2]
```

```
Columns 1 to 2
 1.000000000000000e+00  0.000000000000000e+00
 0.000000000000000e+00  5.551115123125783e-17
--> rank(A)
ans =
  <int32>  - size: [1 1]
 1
--> rank(A,eps/8)
ans =
  <int32>  - size: [1 1]
 2
```

14.35 RCOND Reciprocal Condition Number Estimate

14.35.1 Usage

The `rcond` function is a FreeMat wrapper around LAPACKs function `XGECON`, which estimates the 1-norm condition number (reciprocal). For the details of the algorithm see the LAPACK documentation. The syntax for its use is

```
   x = rcond(A)
```

where `A` is a matrix.

14.35.2 Example

Here is the reciprocal condition number for a random square matrix

```
--> A = rand(30);
--> rcond(A)
ans =
  <double>  - size: [1 1]
 6.631801037021417e-04
```

And here we calculate the same value using the definition of (reciprocal) condition number

```
--> 1/(norm(A,1)*norm(inv(A),1))
ans =
  <double>  - size: [1 1]
 6.505510742818253e-04
```

Note that the values are very similar. LAPACKs `rcond` function is far more efficient than the explicit calculation (which is also used by the `cond` function.

14.36 REPMAT Array Replication Function

14.36.1 Usage

The `repmat` function replicates an array the specified number of times. The source and destination arrays may be multidimensional. There are three distinct syntaxes for the `repmap` function. The first form:

```
y = repmat(x,n)
```

replicates the array `x` on an `n-times-n` tiling, to create a matrix `y` that has `n` times as many rows and columns as `x`. The output `y` will match `x` in all remaining dimensions. The second form is

```
y = repmat(x,m,n)
```

And creates a tiling of `x` with `m` copies of `x` in the row direction, and `n` copies of `x` in the column direction. The final form is the most general

```
y = repmat(x,[m n p...])
```

where the supplied vector indicates the replication factor in each dimension.

14.36.2 Example

Here is an example of using the `repmat` function to replicate a row 5 times. Note that the same effect can be accomplished (although somewhat less efficiently) by a multiplication.

```
--> x = [1 2 3 4]
x =
  <int32>  - size: [1 4]

Columns 1 to 4
 1  2  3  4
--> y = repmat(x,[5,1])
y =
  <int32>  - size: [5 4]

Columns 1 to 4
 1  2  3  4
 1  2  3  4
 1  2  3  4
 1  2  3  4
 1  2  3  4
```

The `repmat` function can also be used to create a matrix of scalars or to provide replication in arbitrary dimensions. Here we use it to replicate a 2D matrix into a 3D volume.

```
--> x = [1 2;3 4]
x =
  <int32>  - size: [2 2]
```

```
Columns 1 to 2
 1  2
 3  4
--> y = repmat(x,[1,1,3])
y =
  <int32>  - size: [2 2 3]
(:,:,1) =

Columns 1 to 2
 1  2
 3  4
(:,:,2) =

Columns 1 to 2
 1  2
 3  4
(:,:,3) =

Columns 1 to 2
 1  2
 3  4
```

14.37 RESHAPE Reshape An Array

14.37.1 Usage

Reshapes an array from one size to another. Two seperate syntaxes are possible. The first syntax specifies the array dimensions as a sequence of scalar dimensions:

```
y = reshape(x,d1,d2,...,dn).
```

The resulting array has the given dimensions, and is filled with the contents of `x`. The type of `y` is the same as `x`. The second syntax specifies the array dimensions as a vector, where each element in the vector specifies a dimension length:

```
y = reshape(x,[d1,d2,...,dn]).
```

This syntax is more convenient for calling `reshape` using a variable for the argument. The `reshape` function requires that the length of `x` equal the product of the `di` values. Note that arrays are stored in column format, which means that elements in `x` are transferred to the new array `y` starting with the first column first element, then proceeding to the last element of the first column, then the first element of the second column, etc.

14.37.2 Example

Here are several examples of the use of `reshape` applied to various arrays. The first example reshapes a row vector into a matrix.

```
--> a = uint8(1:6)
a =
  <uint8>  - size: [1 6]

Columns 1 to 6
 1  2  3  4  5  6
--> reshape(a,2,3)
ans =
  <uint8>  - size: [2 3]

Columns 1 to 3
 1  3  5
 2  4  6
```

The second example reshapes a longer row vector into a volume with two planes.

```
--> a = uint8(1:12)
a =
  <uint8>  - size: [1 12]

Columns 1 to 12
  1   2   3   4   5   6   7   8   9  10  11  12
--> reshape(a,[2,3,2])
ans =
  <uint8>  - size: [2 3 2]
(:,:,1) =

Columns 1 to 3
  1   3   5
  2   4   6
(:,:,2) =

Columns 1 to 3
  7   9  11
  8  10  12
```

The third example reshapes a matrix into another matrix.

```
--> a = [1,6,7;3,4,2]
a =
  <int32>  - size: [2 3]

Columns 1 to 3
 1  6  7
 3  4  2
--> reshape(a,3,2)
ans =
```

```
  <int32>  - size: [3 2]

Columns 1 to 2
 1  4
 3  7
 6  2
```

14.38 SORT Sort

14.38.1 Usage

Sorts an n-dimensional array along the specified dimensional. The first form sorts the array along the first non-singular dimension.

```
  B = sort(A)
```

Alternately, the dimension along which to sort can be explicitly specified

```
  B = sort(A,dim)
```

FreeMat does not support vector arguments for `dim` - if you need `A` to be sorted along multiple dimensions (i.e., row first, then columns), make multiple calls to `sort`. Also, the direction of the sort can be specified using the `mode` argument

```
  B = sort(A,dim,mode)
```

where `mode = 'ascend'` means to sort the data in ascending order (the default), and `mode = 'descend'` means to sort the data into descending order.

When two outputs are requested from `sort`, the indexes are also returned. Thus, for

```
  [B,IX] = sort(A)
  [B,IX] = sort(A,dim)
  [B,IX] = sort(A,dim,mode)
```

an array `IX` of the same size as `A`, where `IX` records the indices of `A` (along the sorting dimension) corresponding to the output array `B`.

Two additional issues worth noting. First, a cell array can be sorted if each cell contains a `string`, in which case the strings are sorted by lexical order. The second issue is that FreeMat uses the same method as MATLAB to sort complex numbers. In particular, a complex number `a` is less than another complex number `b` if `abs(a) < abs(b)`. If the magnitudes are the same then we test the angle of `a`, i.e. `angle(a) < angle(b)`, where `angle(a)` is the phase of `a` between `-pi,pi`.

14.38.2 Example

Here are some examples of sorting on numerical arrays.

```
--> A = int32(10*rand(4,3))
A =
  <int32>  - size: [4 3]
```

```
Columns 1 to 3
 8  3  7
 5  3  8
 6  5  1
 7  3  5
--> [B,IX] = sort(A)
B =
  <int32>  - size: [4 3]

Columns 1 to 3
 5  3  1
 6  3  5
 7  3  7
 8  5  8
IX =
  <int32>  - size: [4 3]

Columns 1 to 3
 2  1  3
 3  2  4
 4  4  1
 1  3  2
--> [B,IX] = sort(A,2)
B =
  <int32>  - size: [4 3]

Columns 1 to 3
 3  7  8
 3  5  8
 1  5  6
 3  5  7
IX =
  <int32>  - size: [4 3]

Columns 1 to 3
 2  3  1
 2  1  3
 3  2  1
 2  3  1
--> [B,IX] = sort(A,1,'descend')
B =
  <int32>  - size: [4 3]

Columns 1 to 3
 8  5  8
```

```
 7  3  7
 6  3  5
 5  3  1
IX =
  <int32>  - size: [4 3]

Columns 1 to 3
 1  3  2
 4  1  1
 3  2  4
 2  4  3
```

Here we sort a cell array of strings.

```
--> a = {'hello','abba','goodbye','jockey','cake'}
a =
  <cell array> - size: [1 5]

Columns 1 to 3
 hello    abba    goodbye

Columns 4 to 5
 jockey    cake
--> b = sort(a)
b =
  <cell array> - size: [1 5]

Columns 1 to 3
 abba    cake    goodbye

Columns 4 to 5
 hello    jockey
```

14.39 SQUEEZE Remove Singleton Dimensions of an Array

14.39.1 Usage

This function removes the singleton dimensions of an array. The syntax for its use is

```
   y = squeeze(x)
```

where `x` is a multidimensional array. Generally speaking, if `x` is of size `d1 x 1 x d2 x ...`, then `squeeze(x)` is of size `d1 x d2 x ...`, i.e., each dimension of `x` that was singular (size 1) is squeezed out.

14.39.2 Example

Here is a many dimensioned, ungainly array, both before and after squeezing;

```
--> x = zeros(1,4,3,1,1,2);
--> size(x)
ans =
  <uint32>  - size: [1 6]

Columns 1 to 6
 1  4  3  1  1  2
--> y = squeeze(x);
--> size(y)
ans =
  <uint32>  - size: [1 3]

Columns 1 to 3
 4  3  2
```

14.40 TRANSPOSE Matrix Transpose

14.40.1 Usage

Performs a (nonconjugate) transpose of a matrix. The syntax for its use is

```
    y = transpose(x)
```

and is a synonym for `y = x.'`.

14.40.2 Example

Here is an example of the transpose of a complex matrix. Note that the entries are not conjugated.

```
--> A = [1+i,2+i;3-2*i,4+2*i]
A =
  <complex>  - size: [2 2]

Columns 1 to 2
   1+  1 i    2+  1 i
   3 -2 i    4+  2 i
--> transpose(A)
ans =
  <complex>  - size: [2 2]

Columns 1 to 2
   1+  1 i    3 -2 i
   2+  1 i    4+  2 i
```

14.41 UNIQUE Unique

14.41.1 Usage

Returns a vector containing the unique elements of an array. The first form is simply

```
y = unique(x)
```

where `x` is either a numerical array or a cell-array of strings. The result is sorted in increasing order. You can also retrieve two sets of index vectors

```
[y, m, n] = unique(x)
```

such that `y = x(m)` and `x = y(n)`. If the argument `x` is a matrix, you can also indicate that FreeMat should look for unique rows in the matrix via

```
y = unique(x,'rows')
```

and

```
[y, m, n] = unique(x,'rows')
```

14.41.2 Example

Here is an example in row mode

```
--> A = randi(1,3*ones(15,3))
A =
  <int32>  - size: [15 3]

Columns 1 to 3
 2  3  2
 2  1  1
 2  2  3
 2  1  3
 2  2  3
 2  1  2
 1  2  2
 1  1  1
 3  1  3
 2  2  2
 1  3  3
 1  2  3
 3  1  1
 3  3  1
 2  3  3
--> unique(A,'rows')
ans =
  <int32>  - size: [14 3]
```

```
Columns 1 to 3
 1  1  1
 1  2  2
 1  2  3
 1  3  3
 2  1  1
 2  1  2
 2  1  3
 2  2  2
 2  2  3
 2  3  2
 2  3  3
 3  1  1
 3  1  3
 3  3  1
--> [b,m,n] = unique(A,'rows');
--> b
ans =
  <int32>  - size: [14 3]

Columns 1 to 3
 1  1  1
 1  2  2
 1  2  3
 1  3  3
 2  1  1
 2  1  2
 2  1  3
 2  2  2
 2  2  3
 2  3  2
 2  3  3
 3  1  1
 3  1  3
 3  3  1
--> A(m,:)
ans =
  <int32>  - size: [14 3]

Columns 1 to 3
 1  1  1
 1  2  2
 1  2  3
 1  3  3
 2  1  1
```

```
 2  1  2
 2  1  3
 2  2  2
 2  2  3
 2  3  2
 2  3  3
 3  1  1
 3  1  3
 3  3  1
--> b(n,:)
ans =
  <int32>  - size: [15 3]

Columns 1 to 3
 2  3  2
 2  1  1
 2  2  3
 2  1  3
 2  2  3
 2  1  2
 1  2  2
 1  1  1
 3  1  3
 2  2  2
 1  3  3
 1  2  3
 3  1  1
 3  3  1
 2  3  3
```

Here is an example in vector mode

```
--> A = randi(1,5*ones(10,1))
A =
  <int32>  - size: [10 1]

Columns 1 to 1
 5
 5
 5
 3
 5
 3
 4
 1
 3
 2
```

```
--> unique(A)
ans =
  <int32>  - size: [5 1]

Columns 1 to 1
 1
 2
 3
 4
 5
--> [b,m,n] = unique(A,'rows');
--> b
ans =
  <int32>  - size: [5 1]

Columns 1 to 1
 1
 2
 3
 4
 5
--> A(m)
ans =
  <int32>  - size: [5 1]

Columns 1 to 1
 1
 2
 3
 4
 5
--> b(n)
ans =
  <int32>  - size: [10 1]

Columns 1 to 1
 5
 5
 5
 3
 5
 3
 4
 1
 3
 2
```

For cell arrays of strings.

```
--> A = {'hi','bye','good','tell','hi','bye'}
A =
  <cell array> - size: [1 6]

Columns 1 to 3
 hi    bye    good

Columns 4 to 6
 tell    hi    bye
--> unique(A)
ans =
  <cell array> - size: [4 1]

Columns 1 to 1
 bye
 good
 hi
 tell
```

14.42 XNRM2 BLAS Norm Calculation

14.42.1 Usage

Calculates the 2-norm of a vector. The syntax for its use is

```
y = xnrm2(A)
```

where `A` is the n-dimensional array to analyze. This form uses the underlying BLAS implementation to compute the 2-norm.

14.43 ZEROS Array of Zeros

14.43.1 Usage

Creates an array of zeros of the specified size. Two seperate syntaxes are possible. The first syntax specifies the array dimensions as a sequence of scalar dimensions:

```
y = zeros(d1,d2,...,dn).
```

The resulting array has the given dimensions, and is filled with all zeros. The type of `y` is `float`, a 32-bit floating point array. To get arrays of other types, use the typecast functions (e.g., `uint8`, `int8`, etc.). An alternative syntax is to use the following notation:

```
y = zeros(d1,d2,...,dn,classname)
```

where `classname` is one of 'double', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'float', 'logical'.

The second syntax specifies the array dimensions as a vector, where each element in the vector specifies a dimension length:

```
y = zeros([d1,d2,...,dn]),
```

or

```
y = zeros([d1,d2,...,dn],classname).
```

This syntax is more convenient for calling `zeros` using a variable for the argument. In both cases, specifying only one dimension results in a square matrix output.

14.43.2 Example

The following examples demonstrate generation of some zero arrays using the first form.

```
--> zeros(2,3,2)
ans =
  <float>  - size: [2 3 2]
(:,:,1) =

Columns 1 to 3
 0  0  0
 0  0  0
(:,:,2) =

Columns 1 to 3
 0  0  0
 0  0  0
--> zeros(1,3)
ans =
  <float>  - size: [1 3]

Columns 1 to 3
 0  0  0
```

The same expressions, using the second form.

```
--> zeros([2,6])
ans =
  <float>  - size: [2 6]

Columns 1 to 6
 0  0  0  0  0  0
 0  0  0  0  0  0
--> zeros([1,3])
```

```
ans =
  <float>  - size: [1 3]

Columns 1 to 3
 0  0  0
```

Finally, an example of using the type casting function `uint16` to generate an array of 16-bit unsigned integers with zero values.

```
--> uint16(zeros(3))
ans =
  <uint16>  - size: [3 3]

Columns 1 to 3
 0  0  0
 0  0  0
 0  0  0
```

Here we use the second syntax where the class of the output is specified explicitly

```
--> zeros(3,'int16')
ans =
  <int16>  - size: [3 3]

Columns 1 to 3
 0  0  0
 0  0  0
 0  0  0
```

Chapter 15

Random Number Generation

15.1 RAND Uniform Random Number Generator

15.1.1 Usage

Creates an array of pseudo-random numbers of the specified size. The numbers are uniformly distributed on `[0,1)`. Two seperate syntaxes are possible. The first syntax specifies the array dimensions as a sequence of scalar dimensions:

```
y = rand(d1,d2,...,dn).
```

The resulting array has the given dimensions, and is filled with random numbers. The type of `y` is `double`, a 64-bit floating point array. To get arrays of other types, use the typecast functions.

The second syntax specifies the array dimensions as a vector, where each element in the vector specifies a dimension length:

```
y = rand([d1,d2,...,dn]).
```

This syntax is more convenient for calling `rand` using a variable for the argument.

Finally, `rand` supports two additional forms that allow you to manipulate the state of the random number generator. The first retrieves the state

```
y = rand('state')
```

which is a 625 length integer vector. The second form sets the state

```
rand('state',y)
```

or alternately, you can reset the random number generator with

```
rand('state',0)
```

15.1.2 Example

The following example demonstrates an example of using the first form of the `rand` function.

```
--> rand(2,2,2)
ans =
  <double>  - size: [2 2 2]
(:,:,1) =

Columns 1 to 2
 0.34781131824756051  0.53132887383000482
 0.02764473155572167  0.99580448434567814
(:,:,2) =

Columns 1 to 2
 0.20792435029277934  0.75968119672921053
 0.49210936348492584  0.33647929575280000
```

The second example demonstrates the second form of the `rand` function.

```
--> rand([2,2,2])
ans =
  <double>  - size: [2 2 2]
(:,:,1) =

Columns 1 to 2
 0.86696050128726199  0.21740218080680240
 0.27140169990196550  0.68970886822238253
(:,:,2) =

Columns 1 to 2
 0.23048304877176773  0.38978704519267782
 0.17207596813829429  0.95446606830622471
```

The third example computes the mean and variance of a large number of uniform random numbers. Recall that the mean should be `1/2`, and the variance should be `1/12 ~ 0.083`.

```
--> x = rand(1,10000);
--> mean(x)
ans =
  <double>  - size: [1 1]
 0.5023477828209344
--> var(x)
ans =
  <double>  - size: [1 1]
 0.08398086618909606
```

Now, we use the state manipulation functions of `rand` to exactly reproduce a random sequence. Note that unlike using `seed`, we can exactly control where the random number generator starts by saving the state.

```
--> rand('state',0)    % restores us to startup conditions
--> a = rand(1,3)      % random sequence 1
a =
  <double>  - size: [1 3]

Columns 1 to 3
 0.375948080123701178  0.018339481363303323  0.913417011246358990
--> b = rand('state'); % capture the state vector
--> c = rand(1,3)      % random sequence 2
c =
  <double>  - size: [1 3]

Columns 1 to 3
 0.35798651897090505  0.76039915395710944  0.80767825652147507
--> rand('state',b);   % restart the random generator so...
--> c = rand(1,3)      % we get random sequence 2 again
c =
  <double>  - size: [1 3]

Columns 1 to 3
 0.35798651897090505  0.76039915395710944  0.80767825652147507
```

15.2 RANDBETA Beta Deviate Random Number Generator

15.2.1 Usage

Creates an array of beta random deviates based on the supplied two parameters. The general syntax for `randbeta` is

```
y = randbeta(alpha, beta)
```

where `alpha` and `beta` are the two parameters of the random deviate. There are three forms for calling `randbeta`. The first uses two vectors `alpha` and `beta` of the same size, in which case the output `y` is the same size as both inputs, and each deviate uses the corresponding values of `alpha` and `beta` from the arguments. In the other forms, either `alpha` or `beta` are scalars.

15.2.2 Function Internals

The probability density function (PDF) of a beta random variable is

$$f(x) = x^(a-1) * (1-x)^(b-1)/B(a,b)$$

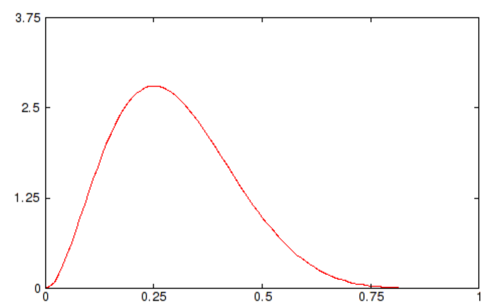
for `x` between 0 and 1. The function `B(a,b)` is defined so that the integral of `f(x)` is 1.

15.2.3 Example

Here is a plot of the PDF of a beta random variable with `a=3`, `b=7`.

```
--> a = 3; b = 7;
--> x = (0:100)/100; t = x.^(a-1).*(1-x).^(b-1);
--> t = t/(sum(t)*.01);
--> plot(x,t);
```

which is plotted as



If we generate a few random deviates with these values, we see they are distributed around the peak of roughly `0.25`.

```
--> randbeta(3*ones(1,5),7*ones(1,5))
ans =
  <float>  - size: [1 5]

Columns 1 to 5
 0.27769583  0.06421967  0.33052161  0.52589989  0.40028515
```

15.3 RANDBIN Generate Binomial Random Variables

15.3.1 Usage

Generates random variables with a binomial distribution. The general syntax for its use is

```
   y = randbin(N,p)
```

where `N` is a vector representing the number of Bernoulli trials, and `p` is the success probability associated with each trial.

15.3.2 Function Internals

A Binomial random variable describes the number of successful outcomes from `N` Bernoulli trials, with the probability of success in each trial being `p`. The probability distribution is

$$P(n) = \frac{N!}{n!(N-n)!}p^n(1-p)^{N-n}$$

15.3.3 Example

Here we generate `10` binomial random variables, corresponding to `N=100` trials, each with probability `p=0.1`, using both `randbin` and then again using `rand` (to simulate the trials):

```
--> randbin(100,.1*ones(1,10))
ans =
  <uint32>  - size: [1 10]

Columns 1 to 10
 13   6   8   9  11   9   6   9   7  10
--> sum(rand(100,10)<0.1)
ans =
  <int32>  - size: [1 10]

Columns 1 to 10
  8  12  10   7  12   4  11   8   9   6
```

15.4 RANDCHI Generate Chi-Square Random Variable

15.4.1 Usage

Generates a vector of chi-square random variables with the given number of degrees of freedom. The general syntax for its use is

```
   y = randchi(n)
```

where `n` is an array containing the degrees of freedom for each generated random variable.

15.4.2 Function Internals

A chi-square random variable is essentially distributed as the squared Euclidean norm of a vector of standard Gaussian random variables. The number of degrees of freedom is generally the number of elements in the vector. In general, the PDF of a chi-square random variable is

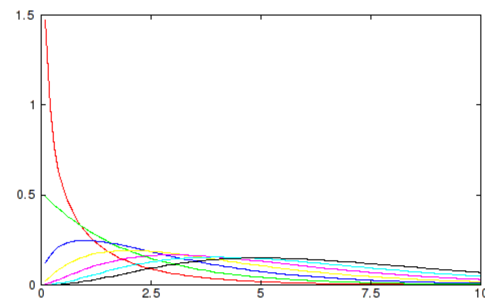
$$f(x) = \frac{x^{r/2-1}e^{-x/2}}{\Gamma(r/2)2^{r/2}}$$

15.4.3 Example

First, a plot of the PDF for a family of chi-square random variables

```
--> f = zeros(7,100);
--> x = (1:100)/10;
--> for n=1:7;t=x.^(n/2-1).*exp(-x/2);f(n,:)=10*t/sum(t);end
--> plot(x,f');
```

The PDF is below:



Here is an example of using **randchi** and **randn** to compute some chi-square random variables with four degrees of freedom.

```
--> randchi(4*ones(1,6))
ans =
  <float>  - size: [1 6]

Columns 1 to 6
 8.9674740  4.0014882  3.2577724  5.5460982  2.5089586  5.7587013
--> sum(randn(4,6).^2)
ans =
  <double>  - size: [1 6]

Columns 1 to 3
  1.1941082197087336  10.6440952477298900   3.6228213108130904

Columns 4 to 6
  8.4424902803814206   2.5030574101217455   1.9057629655510822
```

15.5 RANDEXP Generate Exponential Random Variable

15.5.1 Usage

Generates a vector of exponential random variables with the specified parameter. The general syntax for its use is

```
y = randexp(lambda)
```

where `lambda` is a vector containing the parameters for the generated random variables.

15.5.2 Function Internals

The exponential random variable is usually associated with the waiting time between events in a Poisson random process. The PDF of an exponential random variable is:

$$f(x) = \lambda e^{-\lambda x}$$

15.5.3 Example

Here is an example of using the `randexp` function to generate some exponentially distributed random variables

```
--> randexp(ones(1,6))
ans =
  <float>  - size: [1 6]

Columns 1 to 5
 0.0607746840  0.0018603944  1.1266198158  0.2011622190  0.5079082251

Columns 6 to 6
 3.4205288887
```

15.6 RANDF Generate F-Distributed Random Variable

15.6.1 Usage

Generates random variables with an F-distribution. The general syntax for its use is

```
y = randf(n,m)
```

where `n` and `m` are vectors of the number of degrees of freedom in the numerator and denominator of the chi-square random variables whose ratio defines the statistic.

15.6.2 Function Internals

The statistic `F_{n,m}` is defined as the ratio of two chi-square random variables:

$$F_{n,m} = \frac{\chi_n^2/n}{\chi_m^2/m}$$

The PDF is given by

$$f_{n,m} = \frac{m^{m/2} n^{n/2} x^{n/2-1}}{(m+nx)^{(n+m)/2} B(n/2, m/2)},$$

where `B(a,b)` is the beta function.

15.6.3 Example

Here we use `randf` to generate some F-distributed random variables, and then again using the `randchi` function:

```
--> randf(5*ones(1,9),7)
ans =
  <float>  - size: [1 9]

Columns 1 to 5
 1.194351196  0.906920850  0.755768538  1.502930284  0.062067628

Columns 6 to 9
 1.386031985  1.816088676  0.375541776  3.579419374
--> randchi(5*ones(1,9))./randchi(7*ones(1,9))
ans =
  <float>  - size: [1 9]

Columns 1 to 6
 1.3084840  1.2693102  1.0684280  0.4377135  1.1158004  0.7171145

Columns 7 to 9
 0.4150873  1.8022333  1.4605886
```

15.7 RANDGAMMA Generate Gamma-Distributed Random Variable

15.7.1 Usage

Generates random variables with a gamma distribution. The general syntax for its use is

```
y = randgamma(a,r),
```

where `a` and `r` are vectors describing the parameters of the gamma distribution. Roughly speaking, if `a` is the mean time between changes of a Poisson random process, and we wait for the `r` change, the resulting wait time is Gamma distributed with parameters `a` and `r`.

15.7.2 Function Internals

The Gamma distribution arises in Poisson random processes. It represents the waiting time to the occurance of the `r`-th event in a process with mean time `a` between events. The probability distribution of a Gamma random variable is

$$P(x) = \frac{a^r x^{r-1} e^{-ax}}{\Gamma(r)}.$$

Note also that for integer values of `r` that a Gamma random variable is effectively the sum of `r` exponential random variables with parameter `a`.

15.7.3 Example

Here we use the `randgamma` function to generate Gamma-distributed random variables, and then generate them again using the `randexp` function.

```
--> randgamma(1,15*ones(1,9))
ans =
  <float>  - size: [1 9]

Columns 1 to 6
 22.780443  11.551359  16.853683  12.745702  16.230314  10.744172

Columns 7 to 9
 19.394232  16.361151  17.477238
--> sum(randexp(ones(15,9)))
ans =
  <float>  - size: [1 9]

Columns 1 to 6
 14.6404095  15.1859598  13.3146982  11.4380150   7.2307277  10.8224525

Columns 7 to 9
 14.5270958  12.4630632  11.8753099
```

15.8 RANDI Uniformly Distributed Integer

15.8.1 Usage

Generates an array of uniformly distributed integers between the two supplied limits. The general syntax for `randi` is

```
y = randi(low,high)
```

where `low` and `high` are arrays of integers. Scalars can be used for one of the arguments. The output `y` is a uniformly distributed pseudo-random number between `low` and `high` (inclusive).

15.8.2 Example

Here is an example of a set of random integers between zero and 5:

```
--> randi(zeros(1,6),5*ones(1,6))
ans =
  <int32>  - size: [1 6]

Columns 1 to 6
 1  0  4  1  5  0
```

15.9 RANDMULTI Generate Multinomial-distributed Random Variables

15.9.1 Usage

This function generates samples from a multinomial distribution given the probability of each outcome. The general syntax for its use is

```
   y = randmulti(N,pvec)
```

where `N` is the number of experiments to perform, and `pvec` is the vector of probabilities describing the distribution of outcomes.

15.9.2 Function Internals

A multinomial distribution describes the number of times each of `m` possible outcomes occurs out of `N` trials, where each outcome has a probability `p_i`. More generally, suppose that the probability of a Bernoulli random variable `X_i` is `p_i`, and that

$$\sum_{i=1}^{m} p_i = 1.$$

Then the probability that `X_i` occurs `x_i` times is

$$P_N(x_1, x_2, \ldots, x_n) = \frac{N!}{x_1! \cdots x_n!} p_1^{x_1} \cdots p_n^{x_n}.$$

15.9.3 Example

Suppose an experiment has three possible outcomes, say heads, tails and edge, with probabilities `0.4999`, `0.4999` and `0.0002`, respectively. Then if we perform ten thousand coin flips we get

```
--> randmulti(10000,[0.4999,0.4999,0.0002])
ans =
  <int32>  - size: [1 3]

Columns 1 to 3
 5026  4973     1
```

15.10 RANDN Gaussian (Normal) Random Number Generator

15.10.1 Usage

Creates an array of pseudo-random numbers of the specified size. The numbers are normally distributed with zero mean and a unit standard deviation (i.e., `mu = 0, sigma = 1`). Two seperate syntaxes are possible. The first syntax specifies the array dimensions as a sequence of scalar dimensions:

```
y = randn(d1,d2,...,dn).
```

The resulting array has the given dimensions, and is filled with random numbers. The type of `y` is `double`, a 64-bit floating point array. To get arrays of other types, use the typecast functions.

The second syntax specifies the array dimensions as a vector, where each element in the vector specifies a dimension length:

```
y = randn([d1,d2,...,dn]).
```

This syntax is more convenient for calling `randn` using a variable for the argument.

Finally, `randn` supports two additional forms that allow you to manipulate the state of the random number generator. The first retrieves the state

```
y = randn('state')
```

which is a 625 length integer vector. The second form sets the state

```
randn('state',y)
```

or alternately, you can reset the random number generator with

```
randn('state',0)
```

15.10.2 Function Internals

Recall that the probability density function (PDF) of a normal random variable is

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}.$$

The Gaussian random numbers are generated from pairs of uniform random numbers using a transformation technique.

15.10.3 Example

The following example demonstrates an example of using the first form of the `randn` function.

```
--> randn(2,2,2)
ans =
  <double>  - size: [2 2 2]
(:,:,1) =

Columns 1 to 2
 -1.737491659363370244  -0.566402263751049961
 -0.263398356060278116  -1.011232712469539274
(:,:,2) =

Columns 1 to 2
 -0.402009448685253346   0.055689142534550685
 -1.896556976523083637   0.209817389275412242
```

The second example demonstrates the second form of the `randn` function.

```
--> randn([2,2,2])
ans =
  <double>  - size: [2 2 2]
(:,:,1) =

Columns 1 to 2
 -0.71830955831772469   1.94151925659060609
  0.10096324776995144  -1.17472404899044891
(:,:,2) =

Columns 1 to 2
  0.30478785842683354   3.16847741522331283
 -1.41850940422073712  -0.61299342897060261
```

In the next example, we create a large array of 10000 normally distributed pseudo-random numbers. We then shift the mean to 10, and the variance to 5. We then numerically calculate the mean and variance using `mean` and `var`, respectively.

```
--> x = 10+sqrt(5)*randn(1,10000);
--> mean(x)
ans =
  <double>  - size: [1 1]
 10.013484814146642
--> var(x)
ans =
  <double>  - size: [1 1]
 4.945755069477022
```

Now, we use the state manipulation functions of `randn` to exactly reproduce a random sequence. Note that unlike using `seed`, we can exactly control where the random number generator starts by saving the state.

```
--> randn('state',0)    % restores us to startup conditions
--> a = randn(1,3)      % random sequence 1
a =
  <double>  - size: [1 3]

Columns 1 to 3
 -0.03616399339616805  -0.14041514095502830   0.69338955190756479
--> b = randn('state'); % capture the state vector
--> c = randn(1,3)      % random sequence 2
c =
  <double>  - size: [1 3]

Columns 1 to 3
  0.5997553858968305   0.7086493510746800  -0.9394060974709659
--> randn('state',b);   % restart the random generator so...
--> c = randn(1,3)      % we get random sequence 2 again
c =
  <double>  - size: [1 3]

Columns 1 to 3
  0.5997553858968305   0.7086493510746800  -0.9394060974709659
```

15.11 RANDNBIN Generate Negative Binomial Random Variables

15.11.1 Usage

Generates random variables with a negative binomial distribution. The general syntax for its use is

```
y = randnbin(r,p)
```

where `r` is a vector of integers representing the number of successes, and `p` is the probability of success.

15.11.2 Function Internals

A negative binomial random variable describes the number of failures `x` that occur in `x+r` bernoulli trials, with a success on the `x+r` trial. The pdf is given by

$$P_{r,p}(x) = \binom{x+r-1}{r-1} p^r (1-p)^x.$$

15.11.3 Example

Here we generate some negative binomial random variables:

```
--> randnbin(3*ones(1,4),.01)
ans =
  <uint32>  - size: [1 4]

Columns 1 to 4
 150  274  304  159
--> randnbin(6*ones(1,4),.01)
ans =
  <uint32>  - size: [1 4]

Columns 1 to 4
 657  626  357  663
```

15.12 RANDNCHI Generate Noncentral Chi-Square Random Variable

15.12.1 Usage

Generates a vector of non-central chi-square random variables with the given number of degrees of freedom and the given non-centrality parameters. The general syntax for its use is

```
y = randnchi(n,mu)
```

where `n` is an array containing the degrees of freedom for each generated random variable (with each element of `n` ¿= 1), and `mu` is the non-centrality shift (must be positive).

15.12.2 Function Internals

A non-central chi-square random variable is the sum of a chisquare deviate with `n-1` degrees of freedom plus the square of a normal deviate with mean `mu` and standard deviation 1.

15.12.3 Examples

Here is an example of a non-central chi-square random variable:

```
--> randnchi(5*ones(1,9),0.3)
ans =
  <float>  - size: [1 9]

Columns 1 to 5
 0.1156524420  0.0019547606  0.0028959918  0.0764041170  0.0035292530

Columns 6 to 9
 0.0668950751  0.4731336534  0.0468844920  0.0661755353
```

15.13 RANDNF Generate Noncentral F-Distribution Random Variable

15.13.1 Usage

Generates a vector of non-central F-distributed random variables with the specified parameters. The general syntax for its use is

```
y = randnf(n,m,c)
```

where `n` is the number of degrees of freedom in the numerator, and `m` is the number of degrees of freedom in the denominator. The vector `c` determines the non-centrality shift of the numerator.

15.13.2 Function Internals

A non-central F-distributed random variable is the ratio of a non-central chi-square random variable and a central chi-square random variable, i.e.,

$$F_{n,m,c} = \frac{\chi^2_{n,c}/n}{\chi^2_m/m}.$$

15.13.3 Example

Here we use the `randf` to generate some non-central F-distributed random variables:

```
--> randnf(5*ones(1,9),7,1.34)
ans =
  <float>  - size: [1 9]

Columns 1 to 6
 2.01069760  0.18898647  0.74676013  2.37594628  8.25529575  1.80473185

Columns 7 to 9
 0.22219571  2.26800203  1.96904421
```

15.14 RANDP Generate Poisson Random Variable

15.14.1 Usage

Generates a vector Poisson random variables with the given parameters. The general syntax for its use is

```
y = randp(nu),
```

where `nu` is an array containing the rate parameters for the generated random variables.

15.14.2 Function Internals

A Poisson random variable is generally defined by taking the limit of a binomial distribution as the sample size becomes large, with the expected number of successes being fixed (so that the probability of success decreases as `1/N`). The Poisson distribution is given by

$$P_\nu(n) = \frac{\nu^n e^{-nu}}{n!}.$$

15.14.3 Example

Here is an exmaple of using `randp` to generate some Poisson random variables, and also using `randbin` to do the same using `N=1000` trials to approximate the Poisson result.

```
--> randp(33*ones(1,10))
ans =
  <int32>  - size: [1 10]

Columns 1 to 10
 31  33  34  44  32  29  34  30  32  32
--> randbin(1000*ones(1,10),33/1000*ones(1,10))
ans =
  <uint32>  - size: [1 10]

Columns 1 to 10
 32  36  36  39  33  34  41  33  42  32
```

15.15 SEED Seed the Random Number Generator

15.15.1 Usage

Seeds the random number generator using the given integer seeds. Changing the seed allows you to choose which pseudo-random sequence is generated. The seed takes two `uint32` values:

```
  seed(s,t)
```

where `s` and `t` are the seed values.

15.15.2 Example

Here's an example of how the seed value can be used to reproduce a specific random number sequence.

```
--> seed(32,41);
--> rand(1,5)
ans =
  <double>  - size: [1 5]

Columns 1 to 3
```

```
 0.85888926729303972  0.37271115497527996  0.55512877799859672

Columns 4 to 5
 0.95565654899176766  0.73666959801122378
--> seed(32,41);
--> rand(1,5)
ans =
  <double>  - size: [1 5]

Columns 1 to 3
 0.85888926729303972  0.37271115497527996  0.55512877799859672

Columns 4 to 5
 0.95565654899176766  0.73666959801122378
```

Chapter 16

Input/Ouput Functions

16.1 CSVREAD Read Comma Separated Value (CSV) File

16.1.1 Usage

The `csvread` function reads a text file containing comma separated values (CSV), and returns the resulting numeric matrix (2D). The function supports multiple syntaxes. The first syntax for `csvread` is

```
x = csvread('filename')
```

which attempts to read the entire CSV file into array `x`. The file can contain only numeric values. Each entry in the file should be separated from other entries by a comma. However, FreeMat will attempt to make sense of the entries if the comma is missing (e.g., a space separated file will also parse correctly). For complex values, you must be careful with the spaces). The second form of `csvread` allows you to specify the first row and column (zero-based index)

```
x = csvread('filename',firstrow,firstcol)
```

The last form allows you to specify the range to read also. This form is

```
x = csvread('filename',firstrow,firstcol,readrange)
```

where `readrange` is either a 4-vector of the form `[R1,C1,R2,C2]`, where `R1,C1` is the first row and column to use, and `R2,C2` is the last row and column to use. You can also specify the `readrange` as a spreadsheet range `B12..C34`, in which case the index for the range is 1-based (as in a typical spreadsheet), so that `A1` is the first cell in the upper left corner. Note also that `csvread` is somewhat limited. The number of columns in the file cannot exceed 65535. If it does, bad things will happen.

16.1.2 Example

Here is an example of a CSV file that we wish to read in

```
     sample_data.csv
10, 12, 13, 00, 45, 16
```

```
09, 11, 52, 93, 05, 06
01, 03, 04, 04, 90, -3
14, 17, 13, 67, 30, 43
21, 33, 14, 44, 01, 00
```

We start by reading the entire file

```
--> csvread('sample_data.csv')
ans =
  <int32>  - size: [5 6]

Columns 1 to 6
 10  12  13   0  45  16
  9  11  52  93   5   6
  1   3   4   4  90  -3
 14  17  13  67  30  43
 21  33  14  44   1   0
```

Next, we read everything starting with the second row, and third column

```
--> csvread('sample_data.csv',1,2)
ans =
  <int32>  - size: [4 4]

Columns 1 to 4
 52  93   5   6
  4   4  90  -3
 13  67  30  43
 14  44   1   0
```

Finally, we specify that we only want the `3 x 3` submatrix starting with the second row, and third column

```
--> csvread('sample_data.csv',1,2,[1,2,3,4])
ans =
  <int32>  - size: [2 3]

Columns 1 to 3
 52  93   5
  4   4  90
```

16.2 CSVWRITE Write Comma Separated Value (CSV) File

16.2.1 Usage

The `csvwrite` function writes a given matrix to a text file using comma separated value (CSV) notation. Note that you can create CSV files with arbitrary sized matrices, but that `csvread` has limits on line length. If you need to reliably read and write large matrices, use `rawwrite` and `rawread` respectively. The syntax for `csvwrite` is

```
csvwrite('filename',x)
```

where `x` is a numeric array. The contents of `x` are written to `filename` as comma-separated values. You can also specify a row and column offset to `csvwrite` to force `csvwrite` to write the matrix `x` starting at the specified location in the file. This syntax of the function is

```
csvwrite('filename',x,startrow,startcol)
```

where `startrow` and `startcol` are the offsets in zero-based indexing.

16.2.2 Example

Here we create a simple matrix, and write it to a CSV file

```
--> x = [1,2,3;5,6,7]
x =
  <int32>  - size: [2 3]

Columns 1 to 3
 1  2  3
 5  6  7
--> csvwrite('csvwrite.csv',x)
--> csvread('csvwrite.csv')
ans =
  <int32>  - size: [2 3]

Columns 1 to 3
 1  2  3
 5  6  7
```

Next, we do the same with an offset.

```
--> csvwrite('csvwrite.csv',x,1,2)
--> csvread('csvwrite.csv')
ans =
  <double>  - size: [3 4]

Columns 1 to 4
 0  0  0  0
 0  1  2  3
 0  5  6  7
```

Note the extra zeros.

16.3 DISP Display a Variable or Expression

16.3.1 Usage

Displays the result of a set of expressions. The `disp` function takes a variable number of arguments, each of which is an expression to output:

```
  disp(expr1,expr2,...,exprn)
```

This is functionally equivalent to evaluating each of the expressions without a semicolon after each.

16.3.2 Example

Here are some simple examples of using `disp`.

```
--> a = 32;
--> b = 1:4;
--> disp(a,b,pi)
 32

Columns 1 to 4
 1  2  3  4
 3.141592653589793
```

16.4 FCLOSE File Close Function

16.4.1 Usage

Closes a file handle, or all open file handles. The general syntax for its use is either

```
  fclose(handle)
```

or

```
  fclose('all')
```

In the first case a specific file is closed, In the second, all open files are closed. Note that until a file is closed the file buffers are not flushed. Returns a '0' if the close was successful and a '-1' if the close failed for some reason.

16.4.2 Example

A simple example of a file being opened with `fopen` and then closed with `fclose`.

```
--> fp = fopen('test.dat','wb','ieee-le')
fp =
  <uint32>  - size: [1 1]
 582
--> fclose(fp)
```

```
ans =
  <int32>  - size: [1 1]
 0
```

16.5 FEOF End Of File Function

16.5.1 Usage

Check to see if we are at the end of the file. The usage is

```
  b = feof(handle)
```

The `handle` argument must be a valid and active file handle. The return is true (logical 1) if the current position is at the end of the file, and false (logical 0) otherwise. Note that simply reading to the end of a file will not cause `feof` to return `true`. You must read past the end of the file (which will cause an error anyway). See the example for more details.

16.5.2 Example

Here, we read to the end of the file to demonstrate how `feof` works. At first pass, we force a read of the contents of the file by specifying `inf` for the dimension of the array to read. We then test the end of file, and somewhat counter-intuitively, the answer is `false`. We then attempt to read past the end of the file, which causes an error. An `feof` test now returns the expected value of `true`.

```
--> fp = fopen('test.dat','rb');
--> x = fread(fp,[512,inf],'float');
--> feof(fp)
ans =
  <logical>  - size: [1 1]
 0
--> x = fread(fp,[1,1],'float');
Error: Insufficient data remaining in file to fill out requested size
--> feof(fp)
ans =
  <logical>  - size: [1 1]
 1
```

16.6 FGETLINE Read a String from a File

16.6.1 Usage

Reads a string from a file. The general syntax for its use is

```
  s = fgetline(handle)
```

This function reads characters from the file `handle` into a `string` array `s` until it encounters the end of the file or a newline. The newline, if any, is retained in the output string. If the file is at its end, (i.e., that `feof` would return true on this handle), `fgetline` returns an empty string.

16.6.2 Example

First we write a couple of strings to a test file.

```
--> fp = fopen('testtext','w');
--> fprintf(fp,'String 1\n');
--> fprintf(fp,'String 2\n');
--> fclose(fp);
```

Next, we read then back.

```
--> fp = fopen('testtext','r')
fp =
  <uint32>  - size: [1 1]
 628
--> fgetline(fp)
ans =
  <string>  - size: [1 9]
 String 1

--> fgetline(fp)
ans =
  <string>  - size: [1 9]
 String 2

--> fclose(fp);
```

16.7 FOPEN File Open Function

16.7.1 Usage

Opens a file and returns a handle which can be used for subsequent file manipulations. The general syntax for its use is

```
  fp = fopen(fname,mode,byteorder)
```

Here `fname` is a string containing the name of the file to be opened. `mode` is the mode string for the file open command. The first character of the mode string is one of the following:

- `'r'` Open file for reading. The file pointer is placed at the beginning of the file. The file can be read from, but not written to.
- `'r+'` Open for reading and writing. The file pointer is placed at the beginning of the file. The file can be read from and written to, but must exist at the outset.
- `'w'` Open file for writing. If the file already exists, it is truncated to zero length. Otherwise, a new file is created. The file pointer is placed at the beginning of the file.
- `'w+'` Open for reading and writing. The file is created if it does not exist, otherwise it is truncated to zero length. The file pointer placed at the beginning of the file.

- `'a'` Open for appending (writing at end of file). The file is created if it does not exist. The file pointer is placed at the end of the file.
- `'a+'` Open for reading and appending (writing at end of file). The file is created if it does not exist. The file pointer is placed at the end of the file.

On some platforms (e.g. Win32) it is necessary to add a 'b' for binary files to avoid the operating system's 'CR/LF¡-¿CR' translation.

Finally, FreeMat has the ability to read and write files of any byte-sex (endian). The third (optional) input indicates the byte-endianness of the file. If it is omitted, the native endian-ness of the machine running FreeMat is used. Otherwise, the third argument should be one of the following strings:

- `'le','ieee-le','little-endian','littleEndian','little'`
- `'be','ieee-be','big-endian','bigEndian','big'`

If the file cannot be opened, or the file mode is illegal, then an error occurs. Otherwise, a file handle is returned (which is an integer). This file handle can then be used with `fread`, `fwrite`, or `fclose` for file access.

Note that three handles are assigned at initialization time:

- Handle 0 - is assigned to standard input
- Handle 1 - is assigned to standard output
- Handle 2 - is assigned to standard error

These handles cannot be closed, so that user created file handles start at `3`.

16.7.2 Examples

Here are some examples of how to use `fopen`. First, we create a new file, which we want to be little-endian, regardless of the type of the machine. We also use the `fwrite` function to write some floating point data to the file.

```
--> fp = fopen('test.dat','wb','ieee-le')
fp =
  <uint32>  - size: [1 1]
 570
--> fwrite(fp,float([1.2,4.3,2.1]))
ans =
  <uint32>  - size: [1 1]
 3
--> fclose(fp)
ans =
  <int32>  - size: [1 1]
 0
```

Next, we open the file and read the data back

```
--> fp = fopen('test.dat','rb','ieee-le')
fp =
  <uint32>  - size: [1 1]
 573
--> fread(fp,[1,3],'float')
ans =
  <float>  - size: [1 3]

Columns 1 to 3
 1.2  4.3  2.1
--> fclose(fp)
ans =
  <int32>  - size: [1 1]
 0
```

Now, we re-open the file in append mode and add two additional `float`s to the file.

```
--> fp = fopen('test.dat','a+','le')
fp =
  <uint32>  - size: [1 1]
 576
--> fwrite(fp,float([pi,e]))
ans =
  <uint32>  - size: [1 1]
 2
--> fclose(fp)
ans =
  <int32>  - size: [1 1]
 0
```

Finally, we read all 5 `float` values from the file

```
--> fp = fopen('test.dat','rb','ieee-le')
fp =
  <uint32>  - size: [1 1]
 579
--> fread(fp,[1,5],'float')
ans =
  <float>  - size: [1 5]

Columns 1 to 5
 1.2000000  4.3000002  2.0999999  3.1415927  2.7182817
--> fclose(fp)
ans =
  <int32>  - size: [1 1]
 0
```

16.8 FPRINTF Formated File Output Function (C-Style)

16.8.1 Usage

Prints values to a file. The general syntax for its use is

```
fprintf(fp,format,a1,a2,...).
```

Here `format` is the format string, which is a string that controls the format of the output. The values of the variables `ai` are substituted into the output as required. It is an error if there are not enough variables to satisfy the format string. Note that this `fprintf` command is not vectorized! Each variable must be a scalar. The value `fp` is the file handle. For more details on the format string, see `printf`. Note also that `fprintf` to the file handle `1` is effectively equivalent to `printf`.

16.8.2 Examples

A number of examples are present in the Examples section of the `printf` command.

16.9 FREAD File Read Function

16.9.1 Usage

Reads a block of binary data from the given file handle into a variable of a given shape and precision. The general use of the function is

```
A = fread(handle,size,precision)
```

The `handle` argument must be a valid value returned by the fopen function, and accessable for reading. The `size` argument determines the number of values read from the file. The `size` argument is simply a vector indicating the size of the array `A`. The `size` argument can also contain a single `inf` dimension, indicating that FreeMat should calculate the size of the array along that dimension so as to read as much data as possible from the file (see the examples listed below for more details). The data is stored as columns in the file, not rows.

The third argument determines the type of the data. Legal values for this argument are listed below:

- 'uint8','uchar','unsigned char' for an unsigned, 8-bit integer.
- 'int8','char','integer*1' for a signed, 8-bit integer.
- 'uint16','unsigned short' for an unsigned, 16-bit integer.
- 'int16','short','integer*2' for a signed, 16-bit integer.
- 'uint32','unsigned int' for an unsigned, 32-bit integer.
- 'int32','int','integer*4' for a signed, 32-bit integer.
- 'single','float32','float','real*4' for a 32-bit floating point.

- 'double','float64','real*8' for a 64-bit floating point.
- 'complex','complex*8' for a 64-bit complex floating point (32 bits for the real and imaginary part).
- 'dcomplex','complex*16' for a 128-bit complex floating point (64 bits for the real and imaginary part).

16.9.2 Example

First, we create an array of `512 x 512` Gaussian-distributed `float` random variables, and then writing them to a file called `test.dat`.

```
--> A = float(randn(512));
--> fp = fopen('test.dat','wb');
--> fwrite(fp,A);
--> fclose(fp);
```

Read as many floats as possible into a row vector

```
--> fp = fopen('test.dat','rb');
--> x = fread(fp,[1,inf],'float');
--> who x
  Variable Name      Type   Flags             Size
              x     float                   [1 262144]
```

Read the same floats into a 2-D float array.

```
--> fp = fopen('test.dat','rb');
--> x = fread(fp,[512,inf],'float');
--> who x
  Variable Name      Type   Flags             Size
              x     float                   [512 512]
```

16.10 FSCANF Formatted File Input Function (C-Style)

16.10.1 Usage

Reads values from a file. The general syntax for its use is

```
  [a1,...,an] = fscanf(handle,format)
```

Here `format` is the format string, which is a string that controls the format of the input. Each value that is parsed from the file described by `handle` occupies one output slot. See `printf` for a description of the format. Note that if the file is at the end-of-file, the fscanf will return

16.11 FSEEK Seek File To A Given Position

16.11.1 Usage

Moves the file pointer associated with the given file handle to the specified offset (in bytes). The usage is

```
fseek(handle,offset,style)
```

The `handle` argument must be a value and active file handle. The `offset` parameter indicates the desired seek offset (how much the file pointer is moved in bytes). The `style` parameter determines how the offset is treated. Three values for the `style` parameter are understood:

- string `'bof'` or the value -1, which indicate the seek is relativeto the beginning of the file. This is equivalent to `SEEK_SET` inANSI C.
- string `'cof'` or the value 0, which indicates the seek is relativeto the current position of the file. This is equivalent to `SEEK_CUR` in ANSI C.
- string `'eof'` or the value 1, which indicates the seek is relativeto the end of the file. This is equivalent to `SEEK_END` in ANSIC.

The offset can be positive or negative.

16.11.2 Example

The first example reads a file and then "rewinds" the file pointer by seeking to the beginning. The next example seeks forward by 2048 bytes from the files current position, and then reads a line of 512 floats.

```
--> % First we create the file
--> fp = fopen('test.dat','wb');
--> fwrite(fp,float(rand(4096,1)));
--> fclose(fp);
--> % Now we open it
--> fp = fopen('test.dat','rb');
--> % Read the whole thing
--> x = fread(fp,[1,inf],'float');
--> % Rewind to the beginning
--> fseek(fp,0,'bof');
--> % Read part of the file
--> y = fread(fp,[1,1024],'float');
--> who x y
  Variable Name      Type   Flags             Size
              x     float                    [1 4096]
              y     float                    [1 1024]
--> % Seek 2048 bytes into the file
--> fseek(fp,2048,'cof');
--> % Read 512 floats from the file
```

```
--> x = fread(fp,[512,1],'float');
--> % Close the file
--> fclose(fp);
```

16.12 FTELL File Position Function

16.12.1 Usage

Returns the current file position for a valid file handle. The general use of this function is

```
  n = ftell(handle)
```

The `handle` argument must be a valid and active file handle. The return is the offset into the file relative to the start of the file (in bytes).

16.12.2 Example

Here is an example of using `ftell` to determine the current file position. We read 512 4-byte floats, which results in the file pointer being at position 512*4 = 2048.

```
--> fp = fopen('test.dat','wb');
--> fwrite(fp,randn(512,1));
--> fclose(fp);
--> fp = fopen('test.dat','rb');
--> x = fread(fp,[512,1],'float');
--> ftell(fp)
ans =
  <uint32>  - size: [1 1]
 2048
```

16.13 FWRITE File Write Function

16.13.1 Usage

Writes an array to a given file handle as a block of binary (raw) data. The general use of the function is

```
  n = fwrite(handle,A)
```

The `handle` argument must be a valid value returned by the fopen function, and accessable for writing. The array `A` is written to the file a column at a time. The form of the output data depends on (and is inferred from) the precision of the array `A`. If the write fails (because we ran out of disk space, etc.) then an error is returned. The output `n` indicates the number of elements successfully written.

16.13.2 Example

Heres an example of writing an array of `512 x 512` Gaussian-distributed `float` random variables, and then writing them to a file called `test.dat`.

```
--> A = float(randn(512));
--> fp = fopen('test.dat','wb');
--> fwrite(fp,A);
--> fclose(fp);
```

16.14 GETLINE Get a Line of Input from User

16.14.1 Usage

Reads a line (as a string) from the user. This function has two syntaxes. The first is

```
  a = getline(prompt)
```

where `prompt` is a prompt supplied to the user for the query. The second syntax omits the `prompt` argument:

```
  a = getline
```

Note that this function requires command line input, i.e., it will only operate correctly for programs or scripts written to run inside the FreeMat GUI environment or from the X11 terminal. If you build a stand-alone application and expect it to operate cross-platform, do not use this function (unless you include the FreeMat console in the final application).

16.15 GETPRINTLIMIT Get Limit For Printing Of Arrays

16.15.1 Usage

Returns the limit on how many elements of an array are printed using either the `disp` function or using expressions on the command line without a semi-colon. The default is set to one thousand elements. You can increase or decrease this limit by calling `setprintlimit`. This function is provided primarily so that you can temporarily change the output truncation and then restore it to the previous value (see the examples).

```
   n=getprintlimit
```

where `n` is the current limit in use.

16.15.2 Example

Here is an example of using `getprintlimit` along with `setprintlimit` to temporarily change the output behavior of FreeMat.

```
--> A = randn(100,1);
--> n = getprintlimit
n =
  <uint32>  - size: [1 1]
 1000
--> setprintlimit(5);
--> A
ans =
  <double>  - size: [100 1]

Columns 1 to 1
  0.526461067155618911
 -0.009683349297786812
 -0.088431892510357477
 -0.319293228051011679
 -1.222803614248468396
Print limit has been reached.  Use setprintlimit function to enable longer printouts
--> setprintlimit(n)
```

16.16 INPUT Get Input From User

16.16.1 Usage

The `input` function is used to obtain input from the user. There are two syntaxes for its use. The first is

```
r = input('prompt')
```

in which case, the prompt is presented, and the user is allowed to enter an expression. The expression is evaluated in the current workspace or context (so it can use any defined variables or functions), and returned for assignment to the variable (`r` in this case). In the second form of the `input` function, the syntax is

```
r = input('prompt','s')
```

in which case the text entered by the user is copied verbatim to the output.

16.17 LOAD Load Variables From A File

16.17.1 Usage

Loads a set of variables from a file in a machine independent format. The `load` function takes one argument:

```
load filename,
```

or alternately,

```
load('filename')
```

This command is the companion to `save`. It loads the contents of the file generated by `save` back into the current context. Global and persistent variables are also loaded and flagged appropriately.

16.17.2 Example

Here is a simple example of `save/load`. First, we save some variables to a file.

```
--> D = {1,5,'hello'};
--> s = 'test string';
--> x = randn(512,1);
--> z = zeros(512);
--> who
  Variable Name      Type   Flags             Size
              A     float                    [512 512]
              D      cell                    [1 3]
            ans    double                    [0 0]
             fp    uint32                    [1 1]
              i     int32                    [1 1]
              l      cell                    [1 5]
              n    uint32                    [1 1]
              s    string                    [1 11]
              x    double                    [512 1]
              y     float                    [1 1024]
              z     float                    [512 512]
--> save loadsave.dat
```

Next, we clear all of the variables, and then load them back from the file.

```
--> clear all
--> who
  Variable Name      Type   Flags             Size
--> load loadsave.dat
--> who
  Variable Name      Type   Flags             Size
              A     float                    [512 512]
              D      cell                    [1 3]
            ans    double                    [0 0]
             fp    uint32                    [1 1]
              i     int32                    [1 1]
              l      cell                    [1 5]
              n    uint32                    [1 1]
              s    string                    [1 11]
              x    double                    [512 1]
              y     float                    [1 1024]
              z     float                    [512 512]
```

16.18 PRINTF Formated Output Function (C-Style)

16.18.1 Usage

Prints values to the output. The general syntax for its use is

```
printf(format,a1,a2,...)
```

Here `format` is the format string, which is a string that controls the format of the output. The values of the variables `a_i` are substituted into the output as required. It is an error if there are not enough variables to satisfy the format string. Note that this `printf` command is not vectorized! Each variable must be a scalar.

16.18.2 Format of the format string:

The format string is a character string, beginning and ending in its initial shift state, if any. The format string is composed of zero or more directives: ordinary characters (not unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character conversion specifier. In between there may be (in this order) zero or more flags, an optional minimum field width, and an optional precision.

The arguments must correspond properly (after type promotion) with the conversion specifier, and are used in the order given.

16.18.3 The flag characters:

The character `%` is followed by zero or more of the following flags:

- `\#` The value should be converted to an "alternate form". For `o` conversions, the first character of the output string is made zero (by prefixing a `0` if it was not zero already). For `x` and `X` conversions, a nonzero result has the string `'0x'` (or `'0X'` for `X` conversions) prepended to it. For `a, A, e, E, f, F, g,` and `G` conversions, the result will always contain a decimal point, even if no digits follow it (normally, a decimal point appears in the results of those conversions only if a digit follows). For `g` and `G` conversions, trailing zeros are not removed from the result as they would otherwise be. For other conversions, the result is undefined.

- `0` The value should be zero padded. For `d, i, o, u, x, X, a, A, e, E, f, F, g,` and `G` conversions, the converted value is padded on the left with zeros rather than blanks. If the `0` and `-` flags both appear, the `0` flag is ignored. If a precision is given with a numeric conversion `(d, i, o, u, x, and X)`, the `0` flag is ignored. For other conversions, the behavior is undefined.

- `-` The converted value is to be left adjusted on the field boundary. (The default is right justification.) Except for `n` conversions, the converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A `-` overrides a `0` if both are given.

- `' '` (a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.

- `+` A sign (`+` or `-`) always be placed before a number produced by a signed conversion. By default a sign is used only for negative numbers. A `+` overrides a space if both are used.

16.18.4 The field width:

An optional decimal digit string (with nonzero first digit) specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given). A negative field width is taken as a `'-'` flag followed by a positive field width. In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

16.18.5 The precision:

An optional precision, in the form of a period (`'.'`) followed by an optional decimal digit string. If the precision is given as just `'.'`, or the precision is negative, the precision is taken to be zero. This gives the minimum number of digits to appear for `d, i, o, u, x`, and `X` conversions, the number of digits to appear after the radix character for `a, A, e, E, f`, and `F` conversions, the maximum number of significant digits for `g` and `G` conversions, or the maximum number of characters to be printed from a string for s conversions.

16.18.6 The conversion specifier:

A character that specifies the type of conversion to be applied. The conversion specifiers and their meanings are:

- `d,i` The int argument is converted to signed decimal notation. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is `1`. When `0` is printed with an explicit precision `0`, the output is empty.

- `o,u,x,X` The unsigned int argument is converted to unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal (`x` and `X`) notation. The letters `abcdef` are used for `x` conversions; the letters `ABCDEF` are used for `X` conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is `1`. When `0` is printed with an explicit precision `0`, the output is empty.

- `e,E` The double argument is rounded and converted in the style `[-]d.ddde dd` where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as `6`; if the precision is zero, no decimal-point character appears. An `E` conversion uses the letter `E` (rather than `e`) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is `00`.

- `f,F` The double argument is rounded and converted to decimal notation in the style `[-]ddd.ddd`, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as `6`; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

- `g,G` The double argument is converted in style `f` or `e` (or `F` or `E` for `G` conversions). The precision specifies the number of significant digits. If the precision is missing, `6` digits are given; if the precision is zero, it is treated as `1`. Style e is used if the exponent from its conversion is less than `-4` or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.
- `c` The int argument is converted to an unsigned char, and the resulting character is written.
- `s` The string argument is printed.
- `%` A `'%'` is written. No argument is converted. The complete conversion specification is `'%%'`.

16.18.7 Example

Here are some examples of the use of `printf` with various arguments. First we print out an integer and double value.

```
--> printf('intvalue is %d, floatvalue is %f\n',3,1.53);
intvalue is 3, floatvalue is 1.530000
```

Next, we print out a string value.

```
--> printf('string value is %s\n','hello');
string value is hello
```

Now, we print out an integer using 12 digits, zeros up front.

```
--> printf('integer padded is %012d\n',32);
integer padded is 000000000032
```

Print out a double precision value with a sign, a total of 18 characters (zero prepended if necessary), a decimal point, and 12 digit precision.

```
--> printf('float value is %+018.12f\n',pi);
float value is +0003.141592653590
```

16.19 RAWREAD Read N-dimensional Array From File

16.19.1 Usage

The syntax for `rawread` is

```
function x = rawread(fname,size,precision,byteorder)
```

where `fname` is the name of the file to read from, and `size` is an n-dimensional vector that stores the size of the array in each dimension. The argument `precision` is the type of the data to read in:

- 'uint8','uchar','unsigned char' for unsigned, 8-bit integers

- 'int8','char','integer*1' for signed, 8-bit integers
- 'uint16','unsigned short' for unsigned, 16-bit integers
- 'int16','short','integer*2' for signed, 16-bit integers
- 'uint32','unsigned int' for unsigned, 32-bit integers
- 'int32','int','integer*4' for signed, 32-bit integers
- 'single','float32','float','real*4' for 32-bit floating point
- 'double','float64','real*8' for 64-bit floating point
- 'complex','complex*8' for 64-bit complex floating point (32 bits for the real and imaginary part).
- 'dcomplex','complex*16' for 128-bit complex floating point (64 bits for the real and imaginary part).

As a special feature, one of the size elements can be 'inf', in which case, the largest possible array is read in. If `byteorder` is left unspecified, the file is assumed to be of the same byte-order as the machine `FreeMat` is running on. If you wish to force a particular byte order, specify the `byteorder` argument as

- `'le','ieee-le','little-endian','littleEndian','little'`
- `'be','ieee-be','big-endian','bigEndian','big'`

16.20 SAVE Save Variables To A File

16.20.1 Usage

Saves a set of variables to a file in a machine independent format. There are two formats for the function call. The first is the explicit form, in which a list of variables are provided to write to the file:

```
save filename a1 a2 ...
```

In the second form,

```
save filename
```

all variables in the current context are written to the file. The format of the file is a simple binary encoding (raw) of the data with enough information to restore the variables with the `load` command. The endianness of the machine is encoded in the file, and the resulting file should be portable between machines of similar types (in particular, machines that support IEEE floating point representation).

You can also specify both the filename as a string, in which case you also have to specify the names of the variables to save. In particular

```
save('filename','a1','a2')
```

will save variables `a1` and `a2` to the file.

Starting with version 2.0, FreeMat can also read and write MAT files (the file format used by MATLAB) thanks to substantial work by Thomas Beutlich. Support for MAT files is still in the alpha stages, so please be cautious with using it to store critical data. Also, things like objects wont be saved properly, as will variables that dont exist in MATLAB such as single-precision sparse types. The file format is triggered by the extension. To save files with a MAT format, simply use a filename with a ".mat" ending.

16.20.2 Example

Here is a simple example of `save/load`. First, we save some variables to a file.

```
--> D = {1,5,'hello'};
--> s = 'test string';
--> x = randn(512,1);
--> z = zeros(512);
--> who
  Variable Name      Type   Flags             Size
              A     float                    [512 512]
              D      cell                    [1 3]
            ans     int32                    [1 1]
             fp    uint32                    [1 1]
              i     int32                    [1 1]
              l      cell                    [1 5]
              n    uint32                    [1 1]
              s    string                    [1 11]
              x    double                    [512 1]
              y     float                    [1 1024]
              z     float                    [512 512]
--> save loadsave.dat
```

Next, we clear all of the variables, and then load them back from the file.

```
--> clear all
--> who
  Variable Name      Type   Flags             Size
--> load loadsave.dat
--> who
  Variable Name      Type   Flags             Size
              A     float                    [512 512]
              D      cell                    [1 3]
            ans    double                    [0 0]
             fp    uint32                    [1 1]
              i     int32                    [1 1]
              l      cell                    [1 5]
              n    uint32                    [1 1]
              s    string                    [1 11]
```

```
          x    double                     [512 1]
          y     float                     [1 1024]
          z     float                     [512 512]
```

16.21 SETPRINTLIMIT Set Limit For Printing Of Arrays

16.21.1 Usage

Changes the limit on how many elements of an array are printed using either the `disp` function or using expressions on the command line without a semi-colon. The default is set to one thousand elements. You can increase or decrease this limit by calling

```
  setprintlimit(n)
```

where `n` is the new limit to use.

16.21.2 Example

Setting a smaller print limit avoids pages of output when you forget the semicolon on an expression.

```
--> A = randn(512);
--> setprintlimit(10)
--> A
ans =
  <double>  - size: [512 512]

Columns 1 to 3
 -0.3610410991720306   0.0925734819294048   0.3553025649067262
 -0.5851413047980759   1.2934743129669153  -1.5289721575713926
 -0.7003065867788696  -0.7686479563786363   0.2355850746515537
 -1.5855893795355147
Print limit has been reached.  Use setprintlimit function to enable longer printouts
```

16.22 SPRINTF Formated String Output Function (C-Style)

16.22.1 Usage

Prints values to a string. The general syntax for its use is

```
  y = sprintf(format,a1,a2,...).
```

Here `format` is the format string, which is a string that controls the format of the output. The values of the variables `a_i` are substituted into the output as required. It is an error if there are not enough variables to satisfy the format string. Note that this `sprintf` command is not vectorized! Each variable must be a scalar. The returned value `y` contains the string that would normally have been printed. For more details on the format string, see `printf`.

16.22.2 Examples

Here is an example of a loop that generates a sequence of files based on a template name, and stores them in a cell array.

```
--> l = {}; for i = 1:5; s = sprintf('file_%d.dat',i); l(i) = {s}; end;
--> l
ans =
  <cell array> - size: [1 5]

Columns 1 to 3
 file_1.dat    file_2.dat    file_3.dat

Columns 4 to 5
 file_4.dat    file_5.dat
```

16.23 SSCANF Formated String Input Function (C-Style)

16.23.1 Usage

Reads values from a string. The general syntax for its use is

```
  [a1,...,an] = sscanf(text,format)
```

Here `format` is the format string, which is a string that controls the format of the input. Each value that is parsed from the `text` occupies one output slot. See `printf` for a description of the format.

16.24 STR2NUM Convert a String to a Number

16.24.1 Usage

Converts a string to a number. The general syntax for its use is

```
  x = str2num(string)
```

Here `string` is the data string, which contains the data to be converted into a number. The output is in double precision, and must be typecasted to the appropriate type based on what you need.

Chapter 17

String Functions

17.1 CELLSTR Convert character array to cell array of strings

17.1.1 Usage

The `cellstr` converts a character array matrix into a a cell array of individual strings. Each string in the matrix is placed in a different cell, and extra spaces are removed. The syntax for the command is

```
y = cellstr(x)
```

where `x` is an `N x M` array of characters as a string.

17.1.2 Example

Here is an example of how to use `cellstr`

```
--> a = ['quick';'brown';'fox  ';'is   ']
a =
  <string>  - size: [4 5]
 quick
 brown
 fox
 is
--> cellstr(a)
ans =
  <cell array> - size: [4 1]

Columns 1 to 1
 q
 b
 f
```

```
 i
```

17.2 DEBLANK Remove trailing blanks from a string

17.2.1 Usage

The `deblank` function removes spaces at the end of a string when used with the syntax

```
y = deblank(x)
```

where `x` is a string, in which case, all of the extra spaces in `x` are stripped from the end of the string. Alternately, you can call `deblank` with a cell array of strings

```
y = deblank(c)
```

in which case each string in the cell array is deblanked.

17.2.2 Example

A simple example

```
--> deblank('hello   ')
ans =
  <string>  - size: [1 5]
 hello
```

and a more complex example with a cell array of strings

```
--> deblank({'hello  ','there ','  is  ','  sign  '})
ans =
  <cell array> - size: [1 4]

Columns 1 to 3
 hello    there      is

Columns 4 to 4
   sign
```

17.3 ISALPHA Test for Alpha Characters in a String

17.3.1 Usage

The `isalpha` functions returns a logical array that is 1 for characters in the argument string that are letters, and is a logical 0 for characters in the argument that are not letters. The syntax for its use is

```
x = isalpha(s)
```

where `s` is a `string`. Note that this function is not locale sensitive, and returns a logical 1 for letters in the classic ASCII sense (a through z, and A through Z).

17.3.2 Example

A simple example of `isalpha`:

```
--> isalpha('numb3r5')
ans =
  <logical>  - size: [1 7]

Columns 1 to 7
 1  1  1  1  0  1  0
```

17.4 ISDIGIT Test for Digit Characters in a String

17.4.1 Usage

The `isdigit` functions returns a logical array that is 1 for characters in the argument string that are digits, and is a logical 0 for characters in the argument that are not digits. The syntax for its use is

```
   x = isdigit(s)
```

where `s` is a `string`.

17.4.2 Example

A simple example of `isdigit`:

```
--> isdigit('numb3r5')
ans =
  <logical>  - size: [1 7]

Columns 1 to 7
 0  0  0  0  1  0  1
```

17.5 ISSPACE Test for Space Characters in a String

17.5.1 Usage

The `isspace` functions returns a logical array that is 1 for characters in the argument string that are spaces, and is a logical 0 for characters in the argument that are not spaces. The syntax for its use is

```
   x = isspace(s)
```

where `s` is a `string`. A blank character is considered a space, newline, tab, carriage return, formfeed, and vertical tab.

17.5.2 Example

A simple example of `isspace`:

```
--> isspace('  hello there world ')
ans =
  <logical>  - size: [1 20]

Columns 1 to 19
 1  1  0  0  0  0  0  1  0  0  0  0  0  1  0  0  0  0  0

Columns 20 to 20
 1
```

17.6 STRCMP String Compare Function

17.6.1 USAGE

Compares two strings for equality. The general syntax for its use is

```
  p = strcmp(x,y)
```

where `x` and `y` are two strings. Returns `true` if `x` and `y` are the same size, and are equal (as strings). Otherwise, it returns `false`. In the second form, `strcmp` can be applied to a cell array of strings. The syntax for this form is

```
  p = strcmp(cellstra,cellstrb)
```

where `cellstra` and `cellstrb` are cell arrays of a strings to compare. Also, you can also supply a character matrix as an argument to `strcmp`, in which case it will be converted via `cellstr` (so that trailing spaces are removed), before being compared.

17.6.2 Example

The following piece of code compares two strings:

```
--> x1 = 'astring';
--> x2 = 'bstring';
--> x3 = 'astring';
--> strcmp(x1,x2)
ans =
  <logical>  - size: [1 1]
 0
--> strcmp(x1,x3)
ans =
  <logical>  - size: [1 1]
 1
```

Here we use a cell array strings

```
--> x = {'astring','bstring',43,'astring'}
x =
  <cell array> - size: [1 4]

Columns 1 to 3
 astring    bstring    [43]

Columns 4 to 4
 astring
--> p = strcmp(x,'astring')
p =
  <logical>  - size: [1 4]

Columns 1 to 4
 1  0  0  1
```

Here we compare two cell arrays of strings

```
--> strcmp({'this','is','a','pickle'},{'what','is','to','pickle'})
ans =
  <logical>  - size: [1 4]

Columns 1 to 4
 0  1  0  1
```

Finally, the case where one of the arguments is a matrix string

```
--> strcmp({'this','is','a','pickle'},['peter ';'piper ';'hated ';'pickle']);
```

17.7 STRFIND Find Substring in a String

17.7.1 Usage

Searches through a string for a pattern, and returns the starting positions of the pattern in an array. There are two forms for the `strfind` function. The first is for single strings

```
ndx = strfind(string, pattern)
```

the resulting array `ndx` contains the starting indices in `string` for the pattern `pattern`. The second form takes a cell array of strings

```
ndx = strfind(cells, pattern)
```

and applies the search operation to each string in the cell array.

17.7.2 Example

Here we apply `strfind` to a simple string

```
--> a = 'how now brown cow?'
a =
  <string>  - size: [1 18]
 how now brown cow?
--> b = strfind(a,'ow')
b =
  <int32>  - size: [1 4]

Columns 1 to 4
  2   6  11  16
```

Here we search over multiple strings contained in a cell array.

```
--> a = {'how now brown cow','quick brown fox','coffee anyone?'}
a =
  <cell array> - size: [1 3]

Columns 1 to 3
 how now brown cow    quick brown fox    coffee anyone?
--> b = strfind(a,'ow')
b =
  <cell array> - size: [1 3]

Columns 1 to 3
 [[1 4] int32]    [9]    []
```

17.8 STRNCMP String Compare Function To Length N

17.8.1 USAGE

Compares two strings for equality, but only looks at the first N characters from each string. The general syntax for its use is

```
  p = strncmp(x,y,n)
```

where `x` and `y` are two strings. Returns `true` if `x` and `y` are each at least `n` characters long, and if the first `n` characters from each string are the same. Otherwise, it returns `false`. In the second form, `strncmp` can be applied to a cell array of strings. The syntax for this form is

```
  p = strncmp(cellstra,cellstrb,n)
```

where `cellstra` and `cellstrb` are cell arrays of a strings to compare. Also, you can also supply a character matrix as an argument to `strcmp`, in which case it will be converted via `cellstr` (so that trailing spaces are removed), before being compared.

17.8.2 Example

The following piece of code compares two strings:

```
--> x1 = 'astring';
--> x2 = 'bstring';
--> x3 = 'astring';
--> strncmp(x1,x2,4)
ans =
  <logical>  - size: [1 1]
 0
--> strncmp(x1,x3,4)
ans =
  <logical>  - size: [1 1]
 1
```

Here we use a cell array strings

```
--> x = {'ast','bst',43,'astr'}
x =
  <cell array> - size: [1 4]

Columns 1 to 3
 ast    bst    [43]

Columns 4 to 4
 astr
--> p = strncmp(x,'ast',3)
p =
  <logical>  - size: [1 4]

Columns 1 to 4
 1  0  0  1
```

Here we compare two cell arrays of strings

```
--> strncmp({'this','is','a','pickle'},{'think','is','to','pickle'},3)
ans =
  <logical>  - size: [1 4]

Columns 1 to 4
 1  0  0  1
```

Finally, the case where one of the arguments is a matrix string

```
--> strncmp({'this','is','a','pickle'},['peter ';'piper ';'hated ';'pickle'],4);
```

17.9 STRREP String Replace Function

17.9.1 Usage

Replace every occurance of one string with another. The general syntax for its use is

```
  p = strrep(source,find,replace)
```

Every instance of the string `find` in the string `source` is replaced with the string `replace`. Any of `source`, `find` and `replace` can be a cell array of strings, in which case each entry has the replace operation applied.

17.9.2 Example

Here are some examples of the use of `strrep`. First the case where are the arguments are simple strings

```
--> strrep('Matlab is great','Matlab','FreeMat')
ans =
  <string>  - size: [1 16]
 FreeMat is great
```

And here we have the replace operation for a number of strings:

```
--> strrep({'time is money';'A stitch in time';'No time for games'},'time','money')
ans =
  <cell array> - size: [3 1]

Columns 1 to 1
 money is money
 A stitch in money
 No money for games
```

17.10 STRSTR String Search Function

17.10.1 Usage

Searches for the first occurance of one string inside another. The general syntax for its use is

```
   p = strstr(x,y)
```

where `x` and `y` are two strings. The returned integer `p` indicates the index into the string `x` where the substring `y` occurs. If no instance of `y` is found, then `p` is set to zero.

17.10.2 Example

Some examples of `strstr` in action

```
--> strstr('hello','lo')
ans =
  <int32>  - size: [1 1]
 4
--> strstr('quick brown fox','own')
ans =
```

```
  <int32>  - size: [1 1]
 9
--> strstr('free stuff','lunch')
ans =
  <int32>  - size: [1 1]
 0
```

17.11 STRTRIM Trim Spaces from a String

17.11.1 Usage

Removes the white-spaces at the beginning and end of a string (or a cell array of strings). See `isspace` for a definition of a white-space. There are two forms for the `strtrim` function. The first is for single strings

```
   y = strtrim(strng)
```

where `strng` is a string. The second form operates on a cell array of strings

```
   y = strtrim(cellstr)
```

and trims each string in the cell array.

17.11.2 Example

Here we apply `strtrim` to a simple string

```
--> strtrim('  lot of blank spaces    ');
```

and here we apply it to a cell array

```
--> strtrim({'  space','enough ',' for ',''})
ans =
  <cell array> - size: [1 4]

Columns 1 to 3
 space    enough    for

Columns 4 to 4
 []
```

Chapter 18

Transforms/Decompositions

18.1 EIG Eigendecomposition of a Matrix

18.1.1 Usage

Computes the eigendecomposition of a square matrix. The `eig` function has several forms. The first returns only the eigenvalues of the matrix:

```
s = eig(A)
```

The second form returns both the eigenvectors and eigenvalues as two matrices (the eigenvalues are stored in a diagonal matrix):

```
[V,D] = eig(A)
```

where `D` is the diagonal matrix of eigenvalues, and `V` is the matrix of eigenvectors.

Eigenvalues and eigenvectors for asymmetric matrices `A` normally are computed with balancing applied. Balancing is a scaling step that normaly improves the quality of the eigenvalues and eigenvectors. In some instances (see the Function Internals section for more details) it is necessary to disable balancing. For these cases, two additional forms of `eig` are available:

```
s = eig(A,'nobalance'),
```

which computes the eigenvalues of `A` only, and does not balance the matrix prior to computation. Similarly,

```
[V,D] = eig(A,'nobalance')
```

recovers both the eigenvectors and eigenvalues of `A` without balancing. Note that the 'nobalance' option has no affect on symmetric matrices.

FreeMat also provides the ability to calculate generalized eigenvalues and eigenvectors. Similarly to the regular case, there are two forms for `eig` when computing generalized eigenvector (see the Function Internals section for a description of what a generalized eigenvector is). The first returns only the generalized eigenvalues of the matrix pair `A,B`

```
s = eig(A,B)
```

The second form also computes the generalized eigenvectors, and is accessible via

```
[V,D] = eig(A,B)
```

18.1.2 Function Internals

Recall that `v` is an eigenvector of `A` with associated eigenvalue `d` if

$$Av = dv.$$

This decomposition can be written in matrix form as

$$AV = VD$$

where

$$V = [v_1, v_2, \ldots, v_n], D = \mathrm{diag}(d_1, d_2, \ldots, d_n).$$

The `eig` function uses the `LAPACK` class of functions `GEEVX` to compute the eigenvalue decomposition for non-symmetric (or non-Hermitian) matrices `A`. For symmetric matrices, `SSYEV` and `DSYEV` are used for `float` and `double` matrices (respectively). For Hermitian matrices, `CHEEV` and `ZHEEV` are used for `complex` and `dcomplex` matrices.

For some matrices, the process of balancing (in which the rows and columns of the matrix are pre-scaled to facilitate the search for eigenvalues) is detrimental to the quality of the final solution. This is particularly true if the matrix contains some elements on the order of round off error. See the Example section for an example.

A generalized eigenvector of the matrix pair `A,B` is simply a vector `v` with associated eigenvalue `d` such that

$$Av = dBv,$$

where `B` is a square matrix of the same size as `A`. This decomposition can be written in matrix form as

$$AV = BVD$$

where

$$V = [v_1, v_2, \ldots, v_n], D = \mathrm{diag}(d_1, d_2, \ldots, d_n).$$

For general matrices `A` and `B`, the `GGEV` class of routines are used to compute the generalized eigendecomposition. If howevever, `A` and `B` are both symmetric (or Hermitian, as appropriate), Then FreeMat first attempts to use `SSYGV` and `DSYGV` for `float` and `double` arguments and `CHEGV` and `ZHEGV` for `complex` and `dcomplex` arguments (respectively). These routines requires that `B` also be positive definite, and if it fails to be, FreeMat will revert to the routines used for general arguments.

18.1.3 Example

Some examples of eigenvalue decompositions. First, for a diagonal matrix, the eigenvalues are the diagonal elements of the matrix.

```
--> A = diag([1.02f,3.04f,1.53f])
A =
  <float>  - size: [3 3]

Columns 1 to 3
 1.02  0.00  0.00
 0.00  3.04  0.00
 0.00  0.00  1.53
--> eig(A)
ans =
  <float>  - size: [3 1]

Columns 1 to 1
 1.02
 1.53
 3.04
```

Next, we compute the eigenvalues of an upper triangular matrix, where the eigenvalues are again the diagonal elements.

```
--> A = [1.0f,3.0f,4.0f;0,2.0f,6.7f;0.0f,0.0f,1.0f]
A =
  <float>  - size: [3 3]

Columns 1 to 3
 1.0  3.0  4.0
 0.0  2.0  6.7
 0.0  0.0  1.0
--> eig(A)
ans =
  <float>  - size: [3 1]

Columns 1 to 1
 1
 2
 1
```

Next, we compute the complete eigenvalue decomposition of a random matrix, and then demonstrate the accuracy of the solution

```
--> A = float(randn(2))
A =
  <float>  - size: [2 2]

Columns 1 to 2
  0.23514713   0.30281562
 -0.22035977   0.40320489
```

```
--> [V,D] = eig(A)
V =
  <complex>  - size: [2 2]

Columns 1 to 2
  0.76079118 0.00000000 i   0.76079118-0.00000000 i
  0.21111342+ 0.61370015 i   0.21111342-0.61370015 i
D =
  <complex>  - size: [2 2]

Columns 1 to 2
  0.31917602+ 0.24426939 i   0.00000000 0.00000000 i
  0.00000000 0.00000000 i   0.31917602-0.24426939 i
--> A*V - V*D
ans =
  <complex>  - size: [2 2]

Columns 1 to 1
  0.000000014901161-0.000000014901161 i
  0.000000007450581-0.000000014901161 i

Columns 2 to 2
  0.000000014901161+ 0.000000014901161 i
  0.000000007450581+ 0.000000014901161 i
```

Now, we consider a matrix that requires the nobalance option to compute the eigenvalues and eigenvectors properly. Here is an example from MATLAB's manual.

```
--> B = [3,-2,-.9,2*eps;-2,4,1,-eps;-eps/4,eps/2,-1,0;-.5,-.5,.1,1]
B =
  <double>  - size: [4 4]

Columns 1 to 3
 3.0000000000000000e+00  -2.0000000000000000e+00  -9.0000000000000002e-01
 -2.0000000000000000e+00  4.0000000000000000e+00  1.0000000000000000e+00
 -2.7755575615628914e-17  5.5511151231257827e-17  -1.0000000000000000e+00
 -5.0000000000000000e-01  -5.0000000000000000e-01  1.0000000000000001e-01

Columns 4 to 4
 2.2204460492503131e-16
 -1.1102230246251565e-16
 0.0000000000000000e+00
 1.0000000000000000e+00
--> [VB,DB] = eig(B)
VB =
  <double>  - size: [4 4]
```

```
Columns 1 to 3
 6.1530185559605388e-01  -4.1762246969777006e-01  2.2204460492503155e-16
 -7.8806409946428246e-01  -3.2606977112436158e-01  1.1102230246251583e-16
 -9.2698183372058388e-18  -2.6693858183747784e-18  -1.1839955259674313e-32
 1.8936779969214987e-02  8.4809785824658313e-01  -1.0000000000000000e+00

Columns 4 to 4
 -5.6496654290034469e-02
 4.9717055775230473e-02
 -3.6157858745622118e-01
 -9.2929934225320454e-01
DB =
  <double>  - size: [4 4]

Columns 1 to 3
  5.5615528128088307   0.0000000000000000   0.0000000000000000
  0.0000000000000000   1.4384471871911695   0.0000000000000000
  0.0000000000000000   0.0000000000000000   1.0000000000000004
  0.0000000000000000   0.0000000000000000   0.0000000000000000

Columns 4 to 4
  0.0000000000000000
  0.0000000000000000
  0.0000000000000000
 -1.0000000000000004
--> B*VB - VB*DB
ans =
  <double>  - size: [4 4]

Columns 1 to 3
 0.0000000000000000e+00  1.1102230246251565e-16  4.9303806576313238e-32
 0.0000000000000000e+00  0.0000000000000000e+00  -2.4651903288156619e-32
 -1.2325951644078309e-32  6.6251990086920913e-32  2.3679910519348634e-32
 -2.4980018054066022e-16  -3.7747582837255322e-15  2.2204460492503131e-16

Columns 4 to 4
 4.8572257327350599e-17
 2.6367796834847468e-16
 -1.6653345369377348e-16
 -1.8913667439946296e+00
--> [VN,DN] = eig(B,'nobalance')
VN =
  <double>  - size: [4 4]

Columns 1 to 3
 6.1530185559605388e-01  -4.1762246969777278e-01  4.8234412134015360e-16
```

```
 -7.8806409946428246e-01  -3.2606977112436364e-01  2.5625069969017808e-16
 -1.0578660948728275e-17  -2.9843602070567202e-18  -1.6574241898197358e-18
 1.8936779969214949e-02  8.4809785824658113e-01  -1.0000000000000000e+00

Columns 4 to 4
 -1.5282143569999315e-01
 1.3448286341599405e-01
 -9.7805718847995637e-01
 4.4318216352998034e-02
DN =
  <double>  - size: [4 4]

Columns 1 to 3
  5.5615528128088290   0.0000000000000000   0.0000000000000000
  0.0000000000000000   1.4384471871911686   0.0000000000000000
  0.0000000000000000   0.0000000000000000   1.0000000000000002
  0.0000000000000000   0.0000000000000000   0.0000000000000000

Columns 4 to 4
  0.0000000000000000
  0.0000000000000000
  0.0000000000000000
 -1.0000000000000000
--> B*VN - VN*DN
ans =
  <double>  - size: [4 4]

Columns 1 to 3
 8.8817841970012523e-16  -4.4408920985006262e-16  2.3163392014575750e-16
 -8.8817841970012523e-16  -1.1102230246251565e-16  -8.6571265337077098e-17
 8.5880399191590657e-18  7.6804841211876305e-19  3.3148483796394727e-18
 -5.5511151231257827e-17  2.2204460492503131e-16  -2.2204460492503131e-16

Columns 4 to 4
 1.3877787807814457e-16
 1.3877787807814457e-16
 0.0000000000000000e+00
 -2.7755575615628914e-17
```

18.2 FFT (Inverse) Fast Fourier Transform Function

18.2.1 Usage

Computes the Discrete Fourier Transform (DFT) of a vector using the Fast Fourier Transform technique. The general syntax for its use is

```
y = fft(x,n,d)
```

where `x` is an `n`-dimensional array of numerical type. Integer types are promoted to the `double` type prior to calculation of the DFT. The argument `n` is the length of the FFT, and `d` is the dimension along which to take the DFT. If —n— is larger than the length of `x` along dimension `d`, then `x` is zero-padded (by appending zeros) prior to calculation of the DFT. If `n` is smaller than the length of `x` along the given dimension, then `x` is truncated (by removing elements at the end) to length `n`.

If `d` is omitted, then the DFT is taken along the first non-singleton dimension of `x`. If `n` is omitted, then the DFT length is chosen to match of the length of `x` along dimension `d`.

18.2.2 Function Internals

The output is computed via

$$y(m_1,\ldots,m_{d-1},l,m_{d+1},\ldots,m_p) = \sum_{k=1}^{n} x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p)e^{-\frac{2\pi(k-1)l}{n}}.$$

For the inverse DFT, the calculation is similar, and the arguments have the same meanings as the DFT:

$$y(m_1,\ldots,m_{d-1},l,m_{d+1},\ldots,m_p) = \frac{1}{n}\sum_{k=1}^{n} x(m_1,\ldots,m_{d-1},k,m_{d+1},\ldots,m_p)e^{\frac{2\pi(k-1)l}{n}}.$$

The FFT is computed using the FFTPack library, available from netlib at `http://www.netlib.org`. Generally speaking, the computational cost for a FFT is (in worst case) `O(n^2)`. However, if `n` is composite, and can be factored as

$$n = \prod_{k=1}^{p} m_k,$$

then the DFT can be computed in

$$O(n\sum_{k=1}^{p} m_k)$$

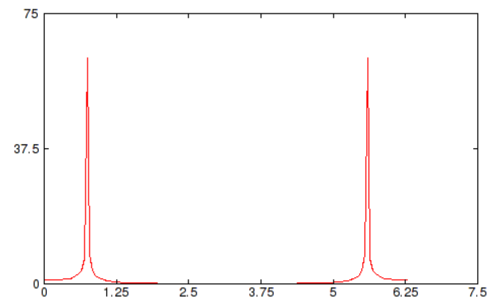
operations. If `n` is a power of 2, then the FFT can be calculated in `O(n log_2 n)`. The calculations for the inverse FFT are identical.

18.2.3 Example

The following piece of code plots the FFT for a sinusoidal signal:

```
--> t = linspace(0,2*pi,128);
--> x = cos(15*t);
--> y = fft(x);
--> plot(t,abs(y));
```

The resulting plot is:



The FFT can also be taken along different dimensions, and with padding and/or truncation. The following example demonstrates the Fourier Transform being computed along each column, and then along each row.

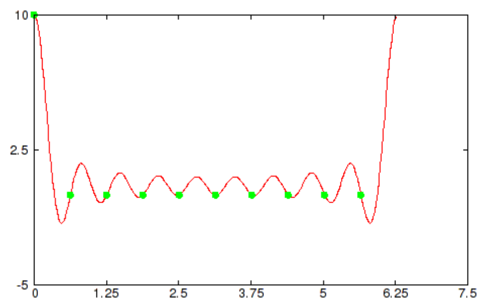
```
--> A = [2,5;3,6]
A =
  <int32>  - size: [2 2]

Columns 1 to 2
 2  5
 3  6
--> real(fft(A,[],1))
ans =
  <float>  - size: [2 2]

Columns 1 to 2
  5  11
 -1  -1
--> real(fft(A,[],2))
ans =
  <float>  - size: [2 2]

Columns 1 to 2
  7  -3
  9  -3
```

Fourier transforms can also be padded using the `n` argument. This pads the signal with zeros prior to taking the Fourier transform. Zero padding in the time domain results in frequency interpolation. The following example demonstrates the FFT of a pulse (consisting of 10 ones) with (red line) and without (green circles) padding.

```
--> delta(1:10) = 1;
--> plot((0:255)/256*pi*2,real(fft(delta,256)),'r-');
```

```
--> hold on
--> plot((0:9)/10*pi*2,real(fft(delta)),'go');
```

The resulting plot is:



18.3 FFTN N-Dimensional Forward FFT

18.3.1 Usage

Computes the DFT of an N-dimensional numerical array along all dimensions. The general syntax for its use is

```
y = fftn(x)
```

which computes the same-size FFTs for each dimension of `x`. Alternately, you can specify the size vector

```
y = fftn(x,dims)
```

where `dims` is a vector of sizes. The array `x` is zero padded or truncated as necessary in each dimension so that the output is of size `dims`. The `fftn` function is implemented by a sequence of calls to `fft`.

18.4 FFTSHIFT Shift FFT Output

18.4.1 Usage

The `fftshift` function shifts the DC component (zero-frequency) of the output from an FFT to the center of the array. For vectors this means swapping the two halves of the vector. For matrices, the first and third quadrants are swapped. So on for N-dimensional arrays. The syntax for its use is

```
y = fftshift(x).
```

Alternately, you can specify that only one dimension be shifted

```
y = fftshift(x,dim).
```

18.5 IFFTN N-Dimensional Inverse FFT

18.5.1 Usage

Computes the inverse DFT of an N-dimensional numerical array along all dimensions. The general syntax for its use is

```
y = ifftn(x)
```

which computes the same-size inverse FFTs for each dimension of `x`. Alternately, you can specify the size vector

```
y = ifftn(x,dims)
```

where `dims` is a vector of sizes. The array `x` is zero padded or truncated as necessary in each dimension so that the output is of size `dims`. The `ifftn` function is implemented by a sequence of calls to `ifft`.

18.6 IFFTSHIFT Inverse Shift FFT Output

18.6.1 Usage

The `ifftshift` function shifts the DC component (zero-frequency) of the output from the center of the array back to the first position and iseffectively the inverse of `fftshift`. For vectors this means swapping the two halves of the vector. For matrices, the first and third quadrants are swapped. So on for N-dimensional arrays. The syntax for its use is

```
y = ifftshift(x).
```

Alternately, you can specify that only one dimension be shifted

```
y = ifftshift(x,dim).
```

18.7 INV Invert Matrix

18.7.1 Usage

Inverts the argument matrix, provided it is square and invertible. The syntax for its use is

```
y = inv(x)
```

Internally, the `inv` function uses the matrix divide operators. For sparse matrices, a sparse matrix solver is used.

18.7.2 Example

Here we invert some simple matrices

```
--> a = randi(zeros(3),5*ones(3))
a =
  <int32>  - size: [3 3]

Columns 1 to 3
 1  1  4
 1  0  1
 0  4  1
--> b = inv(a)
b =
  <float>  - size: [3 3]

Columns 1 to 3
 -0.36363640   1.36363637   0.09090909
 -0.09090909   0.09090909   0.27272728
  0.36363637  -0.36363637  -0.09090909
--> a*b
ans =
  <float>  - size: [3 3]

Columns 1 to 3
  1.000000000000000   0.000000000000000   0.000000000000000
 -0.000000029802322   1.000000000000000   0.000000000000000
  0.000000000000000   0.000000000000000   1.000000000000000
--> b*a
ans =
  <float>  - size: [3 3]

Columns 1 to 3
  1.000000000000000  -0.000000029802322  -0.000000149011612
  0.000000000000000   1.000000000000000   0.000000000000000
  0.000000000000000   0.000000000000000   1.000000000000000
```

18.8 LU LU Decomposition for Matrices

18.8.1 Usage

Computes the LU decomposition for a matrix. The form of the command depends on the type of the argument. For full (non-sparse) matrices, the primary form for `lu` is

```
[L,U,P] = lu(A),
```

where `L` is lower triangular, `U` is upper triangular, and `P` is a permutation matrix such that `L*U = P*A`. The second form is

```
[V,U] = lu(A),
```

where V is P'*L (a row-permuted lower triangular matrix), and U is upper triangular. For sparse, square matrices, the LU decomposition has the following form:

```
[L,U,P,Q,R] = lu(A),
```

where A is a sparse matrix of either double or dcomplex type. The matrices are such that L*U=P*R*A*Q, where L is a lower triangular matrix, U is upper triangular, P and Q are permutation vectors and R is a diagonal matrix of row scaling factors. The decomposition is computed using UMFPACK for sparse matrices, and LAPACK for dense matrices.

18.8.2 Example

First, we compute the LU decomposition of a dense matrix.

```
--> a = float([1,2,3;4,5,8;10,12,3])
a =
  <float>  - size: [3 3]

Columns 1 to 3
  1   2   3
  4   5   8
 10  12   3
--> [l,u,p] = lu(a)
l =
  <float>  - size: [3 3]

Columns 1 to 3
 1.00000000  0.00000000  0.00000000
 0.10000000  1.00000000  0.00000000
 0.40000001  0.24999994  1.00000000
u =
  <float>  - size: [3 3]

Columns 1 to 3
 10.00000000  12.00000000   3.00000000
  0.00000000   0.79999995   2.70000005
  0.00000000   0.00000000   6.12500048
p =
  <float>  - size: [3 3]

Columns 1 to 3
 0  0  1
 1  0  0
 0  1  0
--> l*u
```

```
ans =
  <float>  - size: [3 3]

Columns 1 to 3
 10  12   3
  1   2   3
  4   5   8
--> p*a
ans =
  <float>  - size: [3 3]

Columns 1 to 3
 10  12   3
  1   2   3
  4   5   8
```

Now we repeat the exercise with a sparse matrix, and demonstrate the use of the permutation vectors.

```
--> a = sparse([1,0,0,4;3,2,0,0;0,0,0,1;4,3,2,4])
a =
  <int32>  - size: [4 4]
Matrix is sparse with 9 nonzeros
--> [l,u,p,q,r] = lu(a)
l =
  <double>  - size: [4 4]
Matrix is sparse with 4 nonzeros
u =
  <double>  - size: [4 4]
Matrix is sparse with 9 nonzeros
p =
  <int32>  - size: [1 4]

Columns 1 to 4
 4  2  1  3
q =
  <int32>  - size: [1 4]

Columns 1 to 4
 3  2  1  4
r =
  <double>  - size: [4 4]
Matrix is sparse with 4 nonzeros
--> full(l*a)
ans =
  <double>  - size: [4 4]
```

```
Columns 1 to 4
 1  0  0  4
 3  2  0  0
 0  0  0  1
 4  3  2  4
--> b = r*a
b =
  <double>  - size: [4 4]
Matrix is sparse with 9 nonzeros
--> full(b(p,q))
ans =
  <double>  - size: [4 4]

Columns 1 to 3
 0.15384615384615385  0.23076923076923078  0.30769230769230771
 0.00000000000000000  0.40000000000000002  0.60000000000000009
 0.00000000000000000  0.00000000000000000  0.20000000000000001
 0.00000000000000000  0.00000000000000000  0.00000000000000000

Columns 4 to 4
 0.30769230769230771
 0.00000000000000000
 0.80000000000000004
 1.00000000000000000
```

18.9 QR QR Decomposition of a Matrix

18.9.1 Usage

Computes the QR factorization of a matrix. The `qr` function has multiple forms, with and without pivoting. The non-pivot version has two forms, a compact version and a full-blown decomposition version. The compact version of the decomposition of a matrix of size `M x N` is

```
[q,r] = qr(a,0)
```

where `q` is a matrix of size `M x L` and `r` is a matrix of size `L x N` and `L = min(N,M)`, and `q*r = a`. The QR decomposition is such that the columns of `Q` are orthonormal, and `R` is upper triangular. The decomposition is computed using the LAPACK routine `xgeqrf`, where `x` is the precision of the matrix. Unlike MATLAB (and other MATLAB-compatibles), FreeMat supports decompositions of all four floating point types, `float, complex, double, dcomplex`.

The second form of the non-pivot decomposition omits the second `0` argument:

```
[q,r] = qr(a)
```

This second form differs from the previous form only for matrices with more rows than columns (`M > N`). For these matrices, the full decomposition is of a matrix `Q` of size `M x M` and a matrix `R` of size `M x N`. The full decomposition is computed using the same LAPACK routines as the compact decomposition, but on an augmented matrix `[a 0]`, where enough columns are added to form a square matrix.

Generally, the QR decomposition will not return a matrix `R` with diagonal elements in any specific order. The remaining two forms of the `qr` command utilize permutations of the columns of `a` so that the diagonal elements of `r` are in decreasing magnitude. To trigger this form of the decomposition, a third argument is required, which records the permutation applied to the argument `a`. The compact version is

```
[q,r,e] = qr(a,0)
```

where `e` is an integer vector that describes the permutation of the columns of `a` necessary to reorder the diagonal elements of `r`. This result is computed using the LAPACK routines `(s,d)geqp3`. In the non-compact version of the QR decomposition with pivoting,

```
[q,r,e] = qr(a)
```

the returned matrix `e` is a permutation matrix, such that `q*r*e' = a`.

18.10 SVD Singular Value Decomposition of a Matrix

18.10.1 Usage

Computes the singular value decomposition (SVD) of a matrix. The `svd` function has three forms. The first returns only the singular values of the matrix:

```
s = svd(A)
```

The second form returns both the singular values in a diagonal matrix `S`, as well as the left and right eigenvectors.

```
[U,S,V] = svd(A)
```

The third form returns a more compact decomposition, with the left and right singular vectors corresponding to zero singular values being eliminated. The syntax is

```
[U,S,V] = svd(A,0)
```

18.10.2 Function Internals

Recall that `sigma_i` is a singular value of an `M x N` matrix `A` if there exists two vectors `u_i, v_i` where `u_i` is of length `M`, and `v_i` is of length `u_i` and

$$Av_i = \sigma_i u_i$$

and generally

$$A = \sum_{i=1}^{K} \sigma_i u_i * v_i',$$

where `K` is the rank of `A`. In matrix form, the left singular vectors `u_i` are stored in the matrix `U` as

$$U = [u_1, \ldots, u_m], V = [v_1, \ldots, v_n]$$

The matrix `S` is then of size `M x N` with the singular values along the diagonal. The SVD is computed using the `LAPACK` class of functions `GESDD`.

18.10.3 Examples

Here is an example of a partial and complete singular value decomposition.

```
--> A = float(randn(2,3))
A =
  <float>  - size: [2 3]

Columns 1 to 3
 -1.04076362  -1.61856449   0.55819988
  0.24222484  -1.26936078  -2.91016126
--> [U,S,V] = svd(A)
U =
  <float>  - size: [2 2]

Columns 1 to 2
  0.029025711   0.999578655
  0.999578655  -0.029025711
S =
  <float>  - size: [2 3]

Columns 1 to 3
 3.1849895  0.0000000  0.0000000
 0.0000000  2.0023384  0.0000000
V =
  <float>  - size: [3 3]

Columns 1 to 3
  0.066535234  -0.523066461   0.849690914
 -0.413127303  -0.789596081  -0.453722239
 -0.908239424   0.320841968   0.268628925
--> U*S*V'
ans =
  <float>  - size: [2 3]

Columns 1 to 3
 -1.04076374  -1.61856449   0.55819982
  0.24222499  -1.26936090  -2.91016126
--> svd(A)
ans =
  <float>  - size: [2 1]

Columns 1 to 1
 3.1849895
 2.0023384
```

Chapter 19

Signal Processing Functions

19.1 CONV Convolution Function

19.1.1 Usage

The `conv` function performs a one-dimensional convolution of two vector arguments. The syntax for its use is

```
z = conv(x,y)
```

where `x` and `y` are vectors. The output is of length `nx + ny -1`. The `conv` function calls `conv2` to do the calculation. See its help for more details.

19.2 CONV2 Matrix Convolution

19.2.1 Usage

The `conv2` function performs a two-dimensional convolution of matrix arguments. The syntax for its use is

```
Z = conv2(X,Y)
```

which performs the full 2-D convolution of `X` and `Y`. If the input matrices are of size `[xm,xn]` and `[ym,yn]` respectively, then the output is of size `[xm+ym-1,xn+yn-1]`. Another form is

```
Z = conv2(hcol,hrow,X)
```

where `hcol` and `hrow` are vectors. In this form, `conv2` first convolves `Y` along the columns with `hcol`, and then convolves `Y` along the rows with `hrow`. This is equivalent to `conv2(hcol(:)*hrow(:)',Y)`.

You can also provide an optional `shape` argument to `conv2` via either

```
Z = conv2(X,Y,'shape')
Z = conv2(hcol,hrow,X,'shape')
```

where `shape` is one of the following strings

- `'full'` - compute the full convolution result - this is the default if no `shape` argument is provided.
- `'same'` - returns the central part of the result that is the same size as `X`.
- `'valid'` - returns the portion of the convolution that is computed without the zero-padded edges. In this situation, `Z` has size `[xm-ym+1,xn-yn+1]` when `xm>=ym` and `xn>=yn`. Otherwise`conv2` returns an empty matrix.

19.2.2 Function Internals

The convolution is computed explicitly using the definition:

$$Z(m,n) = \sum_k \sum_j X(k,j)Y(m-k,n-j)$$

If the full output is requested, then `m` ranges over `0 <= m < xm+ym-1` and `n` ranges over `0 <= n < xn+yn-1`. For the case where `shape` is `'same'`, the output ranges over `(ym-1)/2 <= m < xm + (ym-1)/2` and `(yn-1)/2 <= n < xn + (yn-1)/2`.

Chapter 20

Operating System Functions

20.1 CD Change Working Directory Function

20.1.1 Usage

Changes the current working directory to the one specified as the argument. The general syntax for its use is

```
  cd('dirname')
```

but this can also be expressed as

```
  cd 'dirname'
```

or

```
  cd dirname
```

Examples of all three usages are given below. Generally speaking, `dirname` is any string that would be accepted by the underlying OS as a valid directory name. For example, on most systems, `'.'` refers to the current directory, and `'..'` refers to the parent directory. Also, depending on the OS, it may be necessary to "escape" the directory seperators. In particular, if directories are seperated with the backwards-slash character `'\\'`, then the path specification must use double-slashes `'\\\\'`. Note: to get file-name completion to work at this time, you must use one of the first two forms of the command.

20.1.2 Example

The `pwd` command returns the current directory location. First, we use the simplest form of the `cd` command, in which the directory name argument is given unquoted.

```
--> pwd
ans =
  <string>  - size: [1 39]
 /home/basu/Dev/trunk/FreeMat2/build/tmp
```

```
--> pwd
ans =
  <string>  - size: [1 39]
 /home/basu/Dev/trunk/FreeMat2/build/tmp
```

Next, we use the "traditional" form of the function call, using both the parenthesis and a variable to store the quoted string.

```
--> a = pwd;
--> cd(a)
--> pwd
ans =
  <string>  - size: [1 39]
 /home/basu/Dev/trunk/FreeMat2/build/tmp
```

20.2 DIR List Files Function

20.2.1 Usage

An alias for the `ls` function. The general syntax for its use is

```
dir('dirname1','dirname2',...,'dirnameN')
```

but this can also be expressed as

```
dir 'dirname1' 'dirname2' ... 'dirnameN'
```

or

```
dir dirname1 dirname2 ... dirnameN
```

For compatibility with some environments, the function `ls` can also be used instead of `dir`. Generally speaking, `dirname` is any string that would be accepted by the underlying OS as a valid directory name. For example, on most systems, `'.'` refers to the current directory, and `'..'` refers to the parent directory. Two points worth mentioning about the `dir` function:

- To get file-name completion to work at this time, you must use one of the first two forms of the command.
- If you want to capture the output of the `ls` command, use the `system` function instead.

For examples, see the `ls` function.

20.3 GETPATH Get Current Search Path

20.3.1 Usage

Returns a `string` containing the current FreeMat search path. The general syntax for its use is

```
y = getpath
```

The delimiter between the paths depends on the system being used. For Win32, the delimiter is a semicolon. For all other systems, the delimiter is a colon.

20.3.2 Example

The `getpath` function is straightforward.

```
--> getpath
ans =
  <string>  - size: [1 36]
 /home/basu/Dev/trunk/FreeMat2/MFiles
```

20.4 LS List Files Function

20.4.1 Usage

Lists the files in a directory or directories. The general syntax for its use is

```
  ls('dirname1','dirname2',...,'dirnameN')
```

but this can also be expressed as

```
  ls 'dirname1' 'dirname2' ... 'dirnameN'
```

or

```
  ls dirname1 dirname2 ... dirnameN
```

For compatibility with some environments, the function `dir` can also be used instead of `ls`. Generally speaking, `dirname` is any string that would be accepted by the underlying OS as a valid directory name. For example, on most systems, `'.'` refers to the current directory, and `'..'` refers to the parent directory. Also, depending on the OS, it may be necessary to "escape" the directory seperators. In particular, if directories are seperated with the backwards-slash character `'\\'`, then the path specification must use double-slashes `'\\\\'`. Two points worth mentioning about the `ls` function:

- To get file-name completion to work at this time, you must use one of the first two forms of the command.
- If you want to capture the output of the `ls` command, use the `system` function instead.

20.4.2 Example

First, we use the simplest form of the `ls` command, in which the directory name argument is given unquoted.

```
--> ls m*.m
```

Next, we use the "traditional" form of the function call, using both the parenthesis and the quoted string.

```
--> ls('m*.m')
```

In the third version, we use only the quoted string argument without parenthesis.

```
--> ls 'm*.m'
```

20.5 PWD Print Working Directory Function

20.5.1 Usage

Returns a `string` describing the current working directory. The general syntax for its use is

```
  y = pwd
```

20.5.2 Example

The `pwd` function is fairly straightforward.

```
--> pwd
ans =
  <string>  - size: [1 39]
 /home/basu/Dev/trunk/FreeMat2/build/tmp
```

20.6 SETPATH Set Current Search Path

20.6.1 Usage

Changes the current FreeMat search path. The general syntax for its use is

```
  setpath(y)
```

where `y` is a `string` containing a delimited list of directories to be searched for M files and libraries. The delimiter between the paths depends on the system being used. For Win32, the delimiter is a semicolon. For all other systems, the delimiter is a colon.

@Example The `setpath` function is straightforward.

```
--> getpath
ans =
  <string>  - size: [1 36]
 /home/basu/Dev/trunk/FreeMat2/MFiles
--> setpath('/usr/local/FreeMat/MFiles:/localhome/basu/MFiles')
--> getpath
ans =
  <string>  - size: [1 48]
 /usr/local/FreeMat/MFiles:/localhome/basu/MFiles
```

20.7 SYSTEM Call an External Program

20.7.1 Usage

The `system` function allows you to call an external program from within FreeMat, and capture the output. The syntax of the `system` function is

```
  y = system(cmd)
```

where `cmd` is the command to execute. The return array `y` is of type `cell-array`, where each entry in the array corresponds to a line from the output.

20.7.2 Example

Here is an example of calling the `ls` function (the list files function under Un*x-like operating system).

```
--> y = system('ls m*.m')
y =
  <cell array> - size: [0 1]
  []
--> y{1}
```

Chapter 21

Optimization and Curve Fitting

21.1 FITFUN Fit a Function

21.1.1 Usage

Fits `n` (non-linear) functions of `m` variables using least squares and the Levenberg-Marquardt algorithm. The general syntax for its usage is

```
[xopt,yopt] = fitfun(fcn,xinit,y,weights,tol,params...)
```

Where `fcn` is the name of the function to be fit, `xinit` is the initial guess for the solution (required), `y` is the right hand side, i.e., the vector `y` such that:

$$xopt = \arg\min_x \|\text{diag}(weights) * (f(x) - y)\|_2^2,$$

the output `yopt` is the function `fcn` evaluated at `xopt`. The vector `weights` must be the same size as `y`, and contains the relative weight to assign to an error in each output value. Generally, the ith weight should reflect your confidence in the ith measurement. The parameter `tol` is the tolerance used for convergence. The function `fcn` must return a vector of the same size as `y`, and `params` are passed to `fcn` after the argument `x`, i.e.,

$$y = fcn(x, param1, param2, ...).$$

Note that both `x` and `y` (and the output of the function) must all be real variables. Complex variables are not handled yet.

21.2 GAUSFIT Gaussian Curve Fit

21.2.1 Usage

The `gausfit` routine has the following syntax

```
[mu,sigma,dc,gain,yhat] = gausfit(t,y,w,mug,sigmag,dcg,gaing).
```

where the required inputs are

- `t` - the values of the independant variable (e.g., time samples)
- `y` - the values of the dependant variable (e.g., f(t))

The following inputs are all optional, and default values are available for each of them.

- `w` - the weights to use in the fitting (set to ones if omitted)
- `mug` - initial estimate of the mean
- `sigmag` - initial estimate of the sigma (standard deviation)
- `dcg` - initial estimate of the DC value
- `gaing` - initial estimate of the gain

The fit is of the form `yhat=gain*exp((t-mu).^2/(2*sigma^2))+dc`. The outputs are

- `mu` - the mean of the fit
- `sigma` - the sigma of the fit
- `dc` - the dc term of the fit
- `gain` - the gain of the gaussian fit
- `yhat` - the output samples (the Gaussian fits)

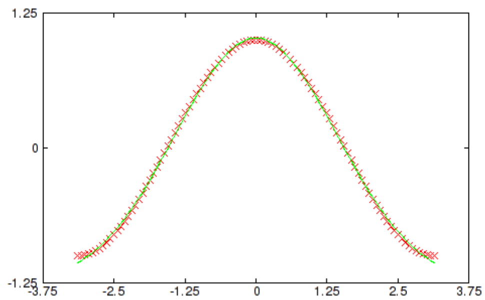
Because the fit is nonlinear, a good initial guess is critical to convergence of the solution. Thus, you can supply initial guesses for each of the parameters using the `mug`, `sigmag`, `dcg`, `gaing` arguments. Any arguments not supplied are estimated using a simple algorithm. In particular, the DC value is estimated by taking the minimum value from the vector `y`. The gain is estimated from the range of `y`. The mean and standard deviation are estimated using the first and second order moments of `y`. This function uses `fitfun`.

21.2.2 Example

Suppose we want to fit a cycle of a cosine using a Gaussian shape.

```
--> t = linspace(-pi,pi);
--> y = cos(t);
--> [mu,sigma,dc,gain,yhat] = gausfit(t,y);
--> plot(t,y,'rx',t,yhat,'g-');
```

Which results in the following plot



21.3 INTERPLIN1 Linear 1-D Interpolation

21.3.1 Usage

Given a set of monotonically increasing `x` coordinates and a corresponding set of `y` values, performs simple linear interpolation to a new set of `x` coordinates. The general syntax for its usage is

```
yi = interplin1(x1,y1,xi)
```

where `x1` and `y1` are vectors of the same length, and the entries in `x1` are monotoniccally increasing. The output vector `yi` is the same size as the input vector `xi`. For each element of `xi`, the values in `y1` are linearly interpolated. For values in `xi` that are outside the range of `x1` the default value returned is `nan`. To change this behavior, you can specify the extrapolation flag:

```
yi = interplin1(x1,y1,xi,extrapflag)
```

Valid options for `extrapflag` are:

- `'nan'` - extrapolated values are tagged with `nan`s
- `'zero'` - extrapolated values are set to zero
- `'endpoint'` - extrapolated values are set to the endpoint values
- `'extrap'` - linear extrapolation is performed

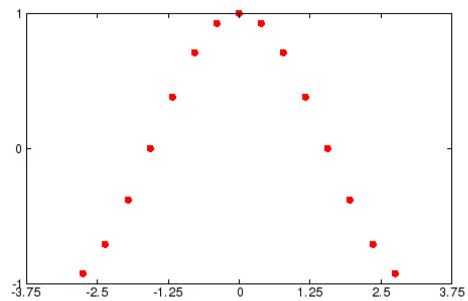
The `x1` and `xi` vectors must be real, although complex types are allowed for `y1`.

21.3.2 Example

Here is an example of simple linear interpolation with the different extrapolation modes. We start with a fairly coarse sampling of a cosine.

```
--> x = linspace(-pi*7/8,pi*7/8,15);
--> y = cos(x);
--> plot(x,y,'ro');
```

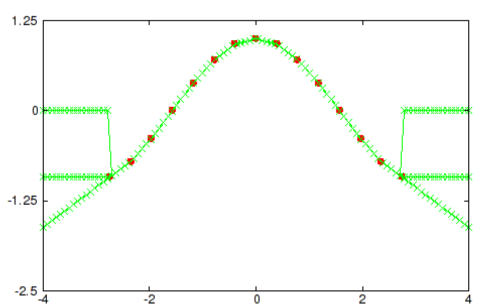
which is shown here



Next, we generate a finer sampling over a slightly broader range (in this case `[-pi,pi]`). First, we demonstrate the `'nan'` extrapolation method

```
--> xi = linspace(-4,4,100);
--> yi_nan = interplin1(x,y,xi,'nan');
--> yi_zero = interplin1(x,y,xi,'zero');
--> yi_endpoint = interplin1(x,y,xi,'endpoint');
--> yi_extrap = interplin1(x,y,xi,'extrap');
--> plot(x,y,'ro',xi,yi_nan,'g-x',xi,yi_zero,'g-x',xi,yi_endpoint,'g-x',xi,yi_extrap,'g-x
```

which is shown here



21.4 POLYFIT Fit Polynomial To Data

21.4.1 Usage

The `polyfit` routine has the following syntax

```
p = polyfit(x,y,n)
```

where `x` and `y` are vectors of the same size, and `n` is the degree of the approximating polynomial. The resulting vector `p` forms the coefficients of the optimal polynomial (in descending degree) that fit `y` with `x`.

21.4.2 Function Internals

The `polyfit` routine finds the approximating polynomial

$$p(x) = p_1 x^n + p_2 x^{n-1} + \cdots + p_n x + p_{n+1}$$

such that

$$\sum_i (p(x_i) - y_i)^2$$

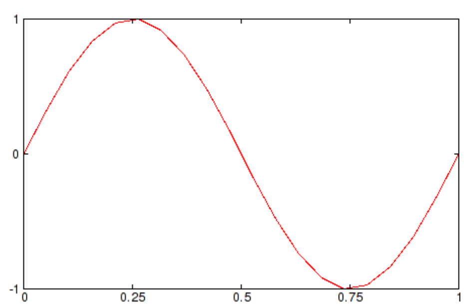
is minimized. It does so by forming the Vandermonde matrix and solving the resulting set of equations using the backslash operator. Note that the Vandermonde matrix can become poorly conditioned with large `n` quite rapidly.

21.4.3 Example

A classic example from Edwards and Penny, consider the problem of approximating a sinusoid with a polynomial. We start with a vector of points evenly spaced on the unit interval, along with a vector of the sine of these points.

```
--> x = linspace(0,1,20);
--> y = sin(2*pi*x);
--> plot(x,y,'r-')
```

The resulting plot is shown here

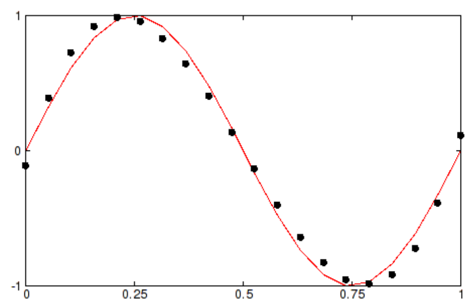


Next, we fit a third degree polynomial to the sine, and use `polyval` to plot it

```
--> p = polyfit(x,y,3)
p =
  <double>  - size: [1 4]
```

```
Columns 1 to 3
  21.91704187823530603  -32.87556281735295727   11.18972672341394770

Columns 4 to 4
  -0.11560289214814741
--> f = polyval(p,x);
--> plot(x,y,'r-',x,f,'ko');
```

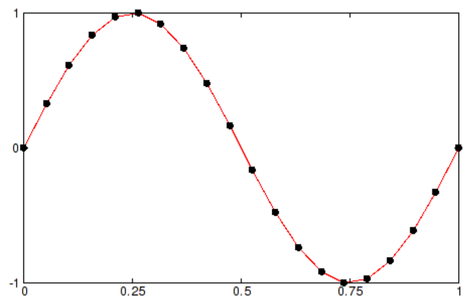
The resulting plot is shown here



Increasing the order improves the fit, as

```
--> p = polyfit(x,y,11)
p =
  <double>  - size: [1 12]

Columns 1 to 3
 1.2464387525545638e+01  -6.8554131391818444e+01  1.3005554675255738e+02

Columns 4 to 6
 -7.1093974942216235e+01  -3.8281376189914866e+01  -1.4122220024211366e+01

Columns 7 to 9
 8.5101772752466474e+01  -5.6416638537956687e-01  -4.1286145173012805e+01

Columns 10 to 12
 -2.9396626046787507e-03  6.2832467411102844e+00  -1.2609031049598294e-09
--> f = polyval(p,x);
--> plot(x,y,'r-',x,f,'ko');
```

The resulting plot is shown here



21.5 POLYVAL Evaluate Polynomial Fit at Selected Points

21.5.1 Usage

The `polyval` routine has the following syntax

```
  y = polyval(p,x)
```

where `p` is a vector of polynomial coefficients, in decreasing degree (as generated by `polyfit`, for example). If `x` is a matrix, the polynomial is evaluated in the matrix sense (in which case `x` must be square).

21.5.2 Function Internals

The polynomial is evaluated using a recursion method. If the polynomial is

$$p(x) = p_1 x^n + p_2 x^{n-1} + \cdots + p_n x + p_{n+1}$$

then the calculation is performed as

$$p(x) = ((p_1)x + p_2)x + p_3$$

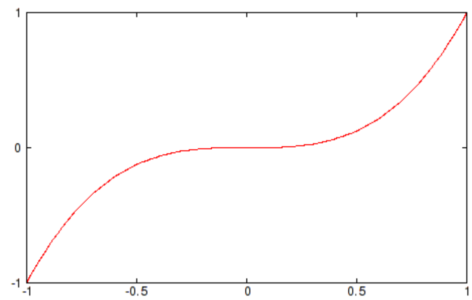
21.5.3 Example

Here is a plot of `x^3` generated using polyval

```
--> p = [1 0 0 0]
p =
  <int32>  - size: [1 4]

Columns 1 to 4
 1  0  0  0
--> x = linspace(-1,1);
--> y = polyval(p,x);
```

```
--> plot(x,y,'r-')
```

Here is the resulting plot



Chapter 22

MPI Functions

22.1 MPIRUN MPI Process Run

22.1.1 Usage

This function is a simple example of how to use FreeMat and MPI to execute functions remotely. More documentation on how to use this function will be written later...

22.2 MPISERVER MPI Process Server

22.2.1 Usage

This function is a simple example of how to use FreeMat and MPI to execute functions remotely. More documentation on how to use this function will be written later...

Chapter 23

Handle-Based Graphics

23.1 AXES Create Handle Axes

23.1.1 Usage

This function has three different syntaxes. The first takes no arguments,

```
h = axes
```

and creates a new set of axes that are parented to the current figure (see `gcf`). The newly created axes are made the current axes (see `gca`) and are added to the end of the list of children for the current figure. The second form takes a set of property names and values

```
h = axes(propertyname,value,propertyname,value,...)
```

Creates a new set of axes, and then sets the specified properties to the given value. This is a shortcut for calling `set(h,propertyname,value)` for each pair. The third form takes a handle as an argument

```
axes(handle)
```

and makes `handle` the current axes, placing it at the head of the list of children for the current figure.

23.2 AXIS Setup Axis Behavior

23.2.1 Usage

Control the axis behavior. There are several versions of the axis command based on what you would like the axis command to do. The first versions set scalings for the current plot. The general syntax for its use is

```
axis([xmin xmax ymin ymax zmin zmax cmin cmax])
```

which sets the limits in the X, Y, Z and color axes. You can also set only the X, Y and Z axes:

```
axis([xmin xmax ymin ymax zmin zmax])
```

or only the X and Y axes:

```
axis([xmin xmax ymin ymax])
```

To retrieve the current axis limits, use the syntax

```
x = axis
```

where `x` is a 4-vector for 2D plots, and a 6-vector for 3D plots.

There are a number of axis options supported by FreeMat. The first version sets the axis limits to be automatically selected by FreeMat for each dimension. This state is the default one for new axes created by FreeMat.

```
axis auto
```

The next option sets all of the axis limits to `manual` mode. This state turns off automatic scaling of the axis based on the children of the current axis object.

```
axis manual
```

The next option sets the axis limits to fit tightly around the data.

```
axis tight
```

The next option adjusts the axis limits and plotbox aspect ratio so that the axis fills the position rectangle.

```
axis fill
```

The next option puts the axis in matrix mode. This mode is equivalent to the standard mode, but with the vertical axis reversed. Thus, the origin of the coordinate system is at the top left corner of the plot. This mode makes plots of matrix elements look normal (i.e., an identity matrix goes from upper left to lower right).

```
axis ij
```

The next option puts the axis in normal mode, with the origin at the lower left corner.

```
axis xy
```

The next option sets the axis parameters (specifically the data aspect ratio) so that equal ticks on each axis represent equal length. In this mode, spheres look spherical insteal of ellipsoidal.

```
axis equal
```

The next option is the same as `axis equal`, but sets the plot box to fit tightly around the data (so no background shows through). It is the best option to use when displaying images.

```
axis image
```

The next option makes the axis box square.

```
axis square
```

The next option restores many of the normal characteristics of the axis. In particular, it undoes the effects of `square image` and `equal` modes.

```
axis normal
```

The next mode freezes axis properties so that 3D objects can be rotated properly.

```
axis vis3d
```

The next mode turns off all labels, tick marks and background.

```
axis on
```

The next mode turns on all labels, tick marks and background.

```
axis off
```

The next mode is similar to `axis off`, but also repacks the figure as tightly as possible. The result is a plot box that takes up the entire `outerposition` vector.

```
axis maximal
```

The `axis` command can also be applied to a particular axis (as opposed to the current axis as returned by `gca`) handle

```
axis(M,...)
```

23.3 AXISPROPERTIES Axis Object Properties

23.3.1 Usage

Below is a summary of the properties for the axis.

- `activepositionproperty` - `four vector` - Not used.
- `alim` - `two vector` - Controls the mapping of transparency. The vector `[a_1,a_2]`@ defines the scale for transparency. Plots then map `a_1` to a completely opaque value, and `a_2` to a completely transparent value. This mapping is applied to the alpha data of the plot data.
- `alimmode` - `{'auto','manual'}` - For `auto` mode, we map the alpha ranges of all objects in the plot to a full scale. For `manual` mode, we use the `alim` vector.
- `ambientlightcolor` - `colorspec` - Not used.
- `box` - `On/Off` - Not used.
- `cameraposition` - `three vector` - Set the position for the camera in axis space.
- `camerapositionmode` - `{'auto','manual'}` - For `manual` mode, the camera position is picked up from the `cameraposition` vector. For `auto` mode, the camera position is set to be centered on the `x` and `y` axis limits, and beyond the `z` maximum limit.

- `cameratarget` - `three vector` - Defines the point in axis space that the camera is targetted at.
- `cameratargetmode` - `{'auto','manual'}` - For `manual` mode the camera target is picked up from the `cameratarget` vector. For `auto` mode, the camera target is chosen to be the center of the three axes.
- `cameraupvector` - `three vector` - Defines the upwards vector for the camera (what is ultimately mapped to the vertical axis of the plot or screen). This vector must not be parallel to the vector that is defined by the optical axis (i.e., the one connecting the target to the camera position).
- `cameraupvectormode` - `{'auto','manual'}` - For `manual` mode, the camera up vector is picked up from the `cameraupvector`. The `auto` mode chooses the up vector to point along the positive `y` axis.
- `cameraviewangle` - `scalar` - Not used.
- `cameraviewanglemode` - `{'auto','manual'}` - Not used.
- `children` - `vector of handles` - A vector containing handles to children of the current axis. Be careful as to how you manipulate this vector. FreeMat uses a reference counting mechanism for graphics objects, so if you remove a handle from the `children` property of an axis, and you have not added it to the `children` property of another object, it will be deleted.
- `clim` - `two vector` - The color range vector. This vector contains two values that dictate how children of this axis get mapped to the colormap. Values between the two endpoints of this vector are mapped to the extremes of the colormap.
- `climmode` - `{'auto','manual'}` - For `auto` mode, the color limits are chosen to span the colordata for all of the children objects. For `manual` mode, the color mapping is based on `clim`.
- `clipping` - `{'on','off'}` - Not used.
- `color` - `colorspec` - The color used to draw the background box for the axes. Defaults to white.
- `colororder` - `color vector` - A vector of color specs (in RGB) that are cycled between when drawing line plots into this axis. The default is order red, green, blue, yellow, magenta, cyan, black.
- `datalimits` - `six vector` - A vector that contains the x, y and z limits of the data for children of the current axis. Changes to this property are ignored - it is calculated by FreeMat based on the datasets.
- `dataaspectratio` - `three vector` - A vector that describes the aspect ratio of the data. You can think of this as the relative scaling of units for each axis. For example, if one unit along the x axis is twice as long as one unit along the y axis, you would specify a data aspect ratio of `[2,1,1]`.

- `dataaspectratiomode` - `{'auto','manual'}` - When the data aspect ratio is set to `manual`, the data is scaled by the data aspect ratio before being plotted. When the data aspect ratio mode is `auto` a complex set of rules are applied to determine how the data should be scaled. If `dataaspectratio` mode is `auto` and `plotboxaspectratio` is `auto`, then the default data aspect ratio is set to `[1,1,1]` and the default plot box aspect ratio is chosen proportional to `[xrange,yrange,zrange]`, where `xrange` is the span of data along the `x` axis, and similarly for `yrange` and `zrange`. If `plotboxaspectratio` is set to `[px,py,pz]`, then the `dataaspectratio` is set to `[xrange/px,yrange/py,zrange/pz]`. If one of the axes has been specified manually, then the data will be scaled to fit the axes as well as possible.
- `fontangle` - `{'normal','italic','oblique'}` - The angle of the fonts used for text labels (e.g., tick labels).
- `fontsize` - `scalar` - The size of fonts used for text labels (tick labels).
- `fontunits` - Not used.
- `fontweight` - `{'normal','bold','light','demi'}` - The weight of the font used for tick labels.
- `gridlinestyle` - `{'-','--',':','-.','none'}` - The line style to use for drawing the grid lines. Defaults to `':'`.
- `handlevisibility` - Not used.
- `hittest` - Not used.
- `interruptible` - Not used.
- `layer` - Not used.
- `linestyleorder` - `linestyle vector` - A vector of linestyles that are cycled through when plotted line series.
- `linewidth` - `scalar` - The width of line used to draw grid lines, axis lines, and other lines.
- `minorgridlinestyle` - `{'-','--',':','-.','none'}` - The line style used for drawing grid lines through minor ticks.
- `nextplot` - `{'add','replace','replacechildren'}` - Controls how the next plot interacts with the axis. If it is set to `'add'` the next plot will be added to the current axis. If it is set to `'replace'` the new plot replaces all of the previous children.
- `outerposition` - `four vector` - Specifies the coordinates of the outermost box that contains the axis relative to the containing figure. This vector is in normalized coordinates and corresponds to the `x, y, width, height` coordinates of the box.
- `parent` - `handle` - The handle for the containing object (a figure).
- `plotboxaspectratio` - `three vector` - Controls the aspect ratio of the plot box. See the entry under `dataaspectratio` for details on how FreeMat uses this vector in combination with the axis limits and the `plotboxaspectratio` to determine how to scale the data.

- `plotboxaspectratiomode` - `{'auto','manual'}` - The plot box aspect ratio mode interacts with the `dataaspectratiomode` and the axis limits.
- `position` - `fourvector` - The normalized coordinates of the plot box space. Should be inside the rectable defined by `outerposition`.
- `projection` - Not used.
- `selected` - Not used.
- `selectionhighlight` - Not used.
- `tag` - A string that can be set to tag the axes with a name.
- `textheight` - `scalar` - This value is set by FreeMat to the height of the current font in pixels.
- `tickdir` - `{'in','out'}` - The direction of ticks. Defaults to `'in'` for 2D plots, and `'out'` for 3D plots if `tickdirmode` is `auto`.
- `tickdirmode` - `{'auto','manual'}` - When set to `'auto'` the `tickdir` defaults to `'in'` for 2D plots, and `'out'` for 3D plots.
- `ticklength` - `two vector` - The first element is the length of the tick in 2D plots, and the second is the length of the tick in the 3D plots. The lengths are described as fractions of the longer dimension (width or height).
- `tightinset` - Not used.
- `title` - `handle` - The handle of the label used to represent the title of the plot.
- `type` - `string` - Takes the value of `'axes'` for objects of the axes type.
- `units` - Not used.
- `userdata` - `array` - An arbitrary array you can set to anything you want.
- `visible` - `{'on','off'}` - If set to `'on'` the axes are drawn as normal. If set to `'off'`, only the children of the axes are drawn. The plot box, axis lines, and tick labels are not drawn.
- `xaxislocation` - `{'top','bottom'}` - Controls placement of the x axis.
- `yaxislocation` - `{'left','right'}` - Controls placement of the y axis.
- `xcolor` - `colorspec` - The color of the x elements including the the x axis line, ticks, grid lines and tick labels
- `ycolor` - `colorspec` - The color of the y elements including the the y axis line, ticks, grid lines and tick labels.
- `zcolor` - `colorspec` - The color of the z elements including the the z axis line, ticks, grid lines and tick labels.

- `xdir` - `{'normal','reverse'}` - For `normal`, axes are drawn as you would expect (e.g, in default 2D mode, the x axis has values increasing from left to right. For `reverse`, the x axis has values increasing from right to left.
- `ydir` - `{'normal','reverse'}` - For `normal`, axes are drawn as you would expect (e.g, in default 2D mode, the y axis has values increasing from bottom to top. For `reverse`, the y axis has values increasing from top to bottom.
- `zdir` - `{'normal','reverse'}` - For `normal`, axes are drawn as you would expect. In default 3D mode, the z axis has values increasing in some direction (usually up). For `reverse` the z axis increases in the opposite direction.
- `xgrid` - `{'on','off'}` - Set to `on` to draw grid lines from ticks on the x axis.
- `ygrid` - `{'on','off'}` - Set to `on` to draw grid lines from ticks on the y axis.
- `zgrid` - `{'on','off'}` - Set to `on` to draw grid lines from ticks on the z axis.
- `xlabel` - `handle` - The handle of the text label attached to the x axis. The position of that label and the rotation angle is computed automatically by FreeMat.
- `ylabel` - `handle` - The handle of the text label attached to the y axis. The position of that label and the rotation angle is computed automatically by FreeMat.
- `zlabel` - `handle` - The handle of the text label attached to the z axis. The position of that label and the rotation angle is computed automatically by FreeMat.
- `xlim` - `two vector` - Contains the limits of the data along the x axis. These are set automatically for `xlimmode`. When manually set it allows you to zoom into the data. The first element of this vector should be the smallest x value you want mapped to the axis, and the second element should be the largest.
- `ylim` - `two vector` - Contains the limits of the data along the y axis. These are set automatically for `ylimmode`. When manually set it allows you to zoom into the data. The first element of this vector should be the smallest y value you want mapped to the axis, and the second element should be the largest.
- `zlim` - `two vector` - Contains the limits of the data along the z axis. These are set automatically for `zlimmode`. When manually set it allows you to zoom into the data. The first element of this vector should be the smallest z value you want mapped to the axis, and the second element should be the largest.
- `xlimmode` - `{'auto','manual'}` - Determines if `xlim` is determined automatically or if it is determined manually. When determined automatically, it is chosen to span the data range (at least).
- `ylimmode` - `{'auto','manual'}` - Determines if `ylim` is determined automatically or if it is determined manually. When determined automatically, it is chosen to span the data range (at least).

- `zlimmode` - `{'auto','manual'}` - Determines if `zlim` is determined automatically or if it is determined manually. When determined automatically, it is chosen to span the data range (at least).
- `xminorgrid` - `{'on','off'}` - Set to `on` to draw grid lines from minor ticks on the x axis.
- `yminorgrid` - `{'on','off'}` - Set to `on` to draw grid lines from minor ticks on the y axis.
- `zminorgrid` - `{'on','off'}` - Set to `on` to draw grid lines from minor ticks on the z axis.
- `xscale` - `{'linear','log'}` - Determines if the data on the x axis is linear or logarithmically scaled.
- `yscale` - `{'linear','log'}` - Determines if the data on the y axis is linear or logarithmically scaled.
- `zscale` - `{'linear','log'}` - Determines if the data on the z axis is linear or logarithmically scaled.
- `xtick` - `vector` - A vector of x coordinates where ticks are placed on the x axis. Setting this vector allows you complete control over the placement of ticks on the axis.
- `ytick` - `vector` - A vector of y coordinates where ticks are placed on the y axis. Setting this vector allows you complete control over the placement of ticks on the axis.
- `ztick` - `vector` - A vector of z coordinates where ticks are placed on the z axis. Setting this vector allows you complete control over the placement of ticks on the axis.
- `xticklabel - string vector` - A string vector, of the form `'string`string—string'— that contains labels to assign to the labels on the axis. If this vector is shorter than `xtick`, then FreeMat will cycle through the elements of this vector to fill out the labels.
- `yticklabel - string vector` - A string vector, of the form `'string`string—string'— that contains labels to assign to the labels on the axis. If this vector is shorter than `ytick`, then FreeMat will cycle through the elements of this vector to fill out the labels.
- `zticklabel - string vector` - A string vector, of the form `'string`string—string'— that contains labels to assign to the labels on the axis. If this vector is shorter than `ztick`, then FreeMat will cycle through the elements of this vector to fill out the labels.
- `xtickmode` - `{'auto','manual'}` - Set to `'auto'` if you want FreeMat to calculate the tick locations. Setting `'xtick'` will cause this property to switch to `'manual'`.
- `ytickmode` - `{'auto','manual'}` - Set to `'auto'` if you want FreeMat to calculate the tick locations. Setting `'ytick'` will cause this property to switch to `'manual'`.
- `ztickmode` - `{'auto','manual'}` - Set to `'auto'` if you want FreeMat to calculate the tick locations. Setting `'ztick'` will cause this property to switch to `'manual'`.
- `xticklabelmode` - `{'auto','manual'}` - Set to `'auto'` if you want FreeMat to set the tick labels. This will be based on the vector `xtick`.

- `yticklabelmode` - `{'auto','manual'}` - Set to `'auto'` if you want FreeMat to set the tick labels. This will be based on the vector `ytick`.
- `zticklabelmode` - `{'auto','manual'}` - Set to `'auto'` if you want FreeMat to set the tick labels. This will be based on the vector `ztick`.

23.4 CLA Clear Current Axis

23.4.1 Usage

Clears the current axes. The syntax for its use is

```
cla
```

23.5 CLF Clear Figure

23.5.1 Usage

This function clears the contents of the current figure. The syntax for its use is

```
clf
```

23.6 CLIM Adjust Color limits of plot

23.6.1 Usage

There are several ways to use `clim` to adjust the color limits of a plot. The various syntaxes are

```
clim
clim([lo,hi])
clim('auto')
clim('manual')
clim('mode')
clim(handle,...)
```

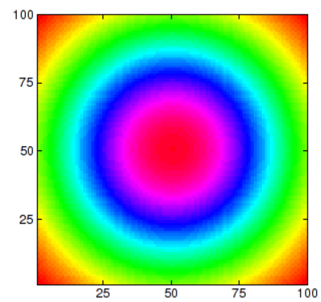
The first form (without arguments), returns a 2-vector containing the current limits. The second form sets the limits on the plot to `[lo,hi]`. The third and fourth form set the mode for the limit to `auto` and `manual` respectively. In `auto` mode, FreeMat chooses the range for the axis automatically. The `clim('mode')` form returns the current mode for the axis (either `'auto'` or `'manual'`). Finally, you can specify the handle of an axis to manipulate instead of using the current one.

23.6.2 Example

Here is an example of using `clim` to change the effective window and level onto an image. First, the image with default limits

```
--> x = repmat(linspace(-1,1),[100,1]); y = x';
--> z = exp(-x.^2-y.^2);
--> image(z)
ans =
  <uint32>  - size: [1 1]
 100002
```

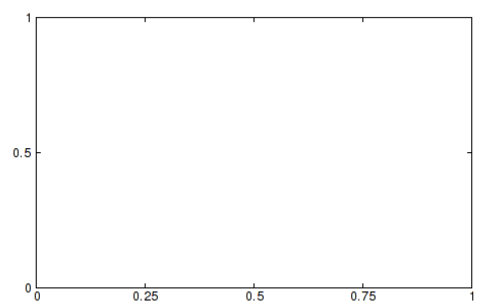
which results in



Next, we change the colorscale of the image using the `clim` function

```
--> clim([0,0.2])
```

which results in



23.7 CLOSE Close Figure Window

23.7.1 Usage

Closes a figure window, either the currently active window, a window with a specific handle, or all figure windows. The general syntax for its use is

```
close(handle)
```

in which case the figure window with the speicified `handle` is closed. Alternately, issuing the command with no argument

```
close
```

is equivalent to closing the currently active figure window. Finally the command

```
close('all')
```

closes all figure windows currently open.

23.8 COLORBAR Add Colorbar to Current Plot

23.8.1 Usage

There are a number of syntaxes for the `colorbar` command. The first takes no arguments, and adds a vertical colorbar to the right of the current axes.

```
colorbar
```

You can also pass properties to the newly created axes object using the second syntax for colorbar

```
colorbar(properties...)
```

23.9 COLORMAP Image Colormap Function

23.9.1 Usage

Changes the colormap for the current figure. The generic syntax for its use is

```
colormap(map)
```

where `map` is a an array organized as `3 \times N`), which defines the RGB (Red Green Blue) coordinates for each color in the colormap. You can also use the function with no arguments to recover the current colormap

```
map = colormap
```

23.9.2 Function Internals

Assuming that the contents of the colormap function argument `c` are labeled as:

$$c = \begin{bmatrix} r_1 & g_1 & b_1 \\ r_1 & g_2 & b_2 \\ r_1 & g_3 & b_3 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

then these columns for the RGB coordinates of pixel in the mapped image. Assume that the image occupies the range $[a, b]$. Then the RGB color of each pixel depends on the value x via the following integer

$$k = 1 + \lfloor 256 \frac{x-a}{b-a} \rfloor,$$

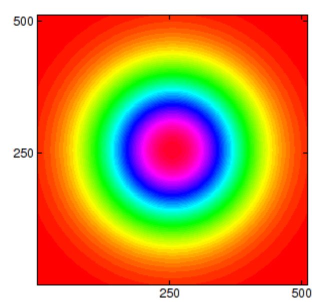
so that a pixel corresponding to image value x will receive RGB color $[r_k, g_k, b_k]$. Colormaps are generally used to pseudo color images to enhance visibility of features, etc.

23.9.3 Examples

We start by creating a smoothly varying image of a 2D Gaussian pulse.

```
--> x = linspace(-1,1,512)'*ones(1,512);
--> y = x';
--> Z = exp(-(x.^2+y.^2)/0.3);
--> image(Z);
```

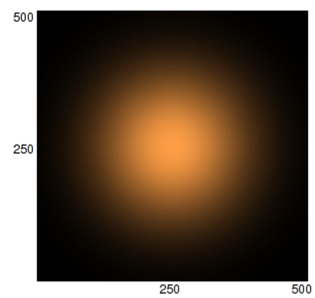
which we display with the default (grayscale) colormap here.



Next we switch to the `copper` colormap, and redisplay the image.

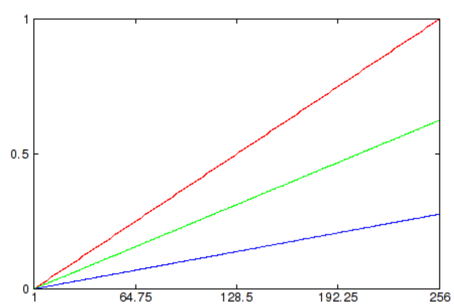
```
--> colormap(copper);
--> image(Z);
```

which results in the following image.



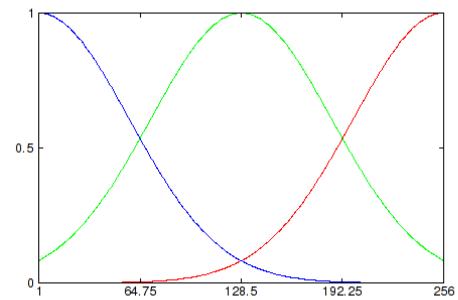
If we capture the output of the `copper` command and plot it, we obtain the following result:

```
--> a = copper;
--> plot(a);
```



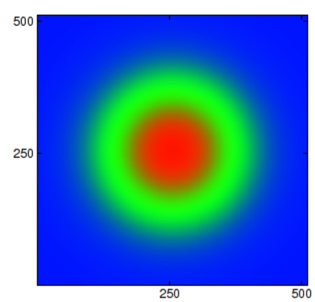
Note that in the output that each of the color components are linear functions of the index, with the ratio between the red, blue and green components remaining constant as a function of index. The result is an intensity map with a copper tint. We can similarly construct a colormap of our own by defining the three components seperately. For example, suppose we take three gaussian curves, one for each color, centered on different parts of the index space:

```
--> t = linspace(0,1,256);
--> A = [exp(-(t-1.0).^2/0.1);exp(-(t-0.5).^2/0.1);exp(-t.^2/0.1)]';
--> plot(A);
```



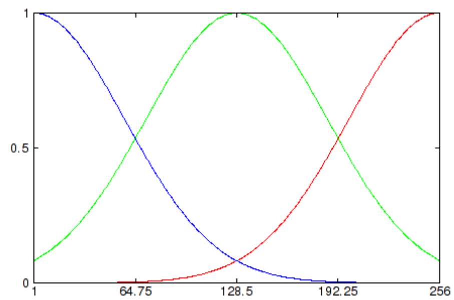
The resulting image has dark bands in it near the color transitions.

```
--> image(Z);
--> colormap(A);
```



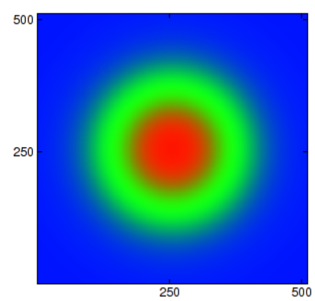
These dark bands are a result of the nonuniform color intensity, which we can correct for by renormalizing each color to have the same norm.

```
--> w = sqrt(sum(A'.^2));
--> sA = diag(1./w)*A;
--> plot(A);
```



The resulting image has no more dark bands.

```
--> image(Z);
--> colormap(A);
```



23.10 COLORSPEC Color Property Description

23.10.1 Usage

There are a number of ways of specifying a color value for a color-based property. Examples include line colors, marker colors, and the like. One option is to specify color as an RGB triplet

```
set(h,'color',[r,g,b])
```

where `r,g,b` are between @[0,1]@. Alternately, you can use color names to specify a color.

- `'none'` - No color.
- `'y','yellow'` - The color @[1,1,0]@ in RGB space.
- `'m','magenta'` - The color @[1,0,1]@ in RGB space.

- 'c','cyan' - The color @[0,1,1]@ in RGB space.
- 'r','red' - The color @[1,0,0]@ in RGB space.
- 'g','green' - The color @[0,1,0]@ in RGB space.
- 'b','blue' - The color @[0,0,1]@ in RGB space.
- 'w','white' - The color @[1,1,1]@ in RGB space.
- 'k','black' - The color @[0,0,0]@ in RGB space.

23.11 COPPER Copper Colormap

23.11.1 Usage

Returns a copper colormap. The syntax for its use is

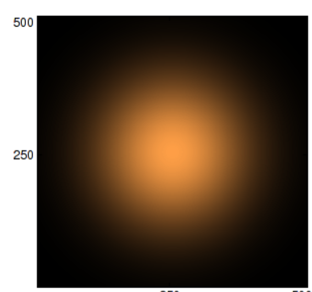
```
y = copper
```

23.11.2 Example

Here is an example of an image displayed with the copper colormap

```
--> x = linspace(-1,1,512)'*ones(1,512);
--> y = x';
--> Z = exp(-(x.^2+y.^2)/0.3);
--> image(Z);
--> colormap(copper);
```

which results in the following image



23.12 COPY Copy Figure Window

23.12.1 Usage

Copies the currently active figure window to the clipboard. The syntax for its use is:

```
copy
```

The resulting figure is copied as a bitmap to the clipboard, and can then be pasted into any suitable application.

23.13 FIGURE Figure Window Select and Create Function

23.13.1 Usage

Changes the active figure window to the specified handle (or figure number). The general syntax for its use is

```
figure(handle)
```

where `handle` is the handle to use. If the figure window corresponding to `handle` does not already exist, a new window with this handle number is created. If it does exist then it is brought to the forefront and made active.

23.14 FIGUREPROPERTIES Figure Object Properties

23.14.1 Usage

Below is a summary of the properties for the axis.

- `alphamap` - `vector` - Contains the alpha (transparency) map for the figure. If this is set to a scalar, then all values are mapped to the same transparency. It defaults to `1`, which is all values being fully opaque. If you set this to a vector, the values of graphics objects will be mapped to different transparency values, based on the setting of their `alphadatamapping` property.
- `color` - `colorspec` - The background color of the figure (defaults to a gray `[0.6,0.6,0.6]`). During printing, this color is set to white, and then is restored.
- `colormap` - `color vector` - an `N x 3` matrix of RGB values that specifies the colormap for the figure. Defaults to an `HSV` map.
- `children` - `handle vector` - the handles for objects that are children of this figure. These should be axis objects.
- `currentaxes` - `handle` - the handle for the current axes. Also returned by `gca`.
- `parent` - Not used.
- `position` - Not used.

- `type` - `string` - returns the string `'figure'`.
- `userdata` - `array` - arbitrary array you can use to store data associated with the figure.
- `nextplot` - `{'add','replace','replacechildren'}` - If set to `'add'` then additional axes are added to the list of children for the current figure. If set to `'replace'`, then a new axis replaces all of the existing children.
- `figsize` - `two vector` - the size of the figure window in pixels (width x height).
- `renderer` - `{'painters','opengl'}` - When set to `'painters'` drawing is based on the Qt drawing methods (which can handle flat shading of surfaces with transparency). If you set the renderer to `'opengl'` then OpenGL is used for rendering. Support for OpenGL is currently in the alpha stage, and FreeMat does not enable it automatically. You can set the renderer mode to `'opengl'` manually to experiment. Also, OpenGL figures cannot be printed yet.

23.15 GCA Get Current Axis

23.15.1 Usage

Returns the handle for the current axis. The syntax for its use is

```
handle = gca
```

where `handle` is the handle of the active axis. All object creation functions will be children of this axis.

23.16 GCF Get Current Figure

23.16.1 Usage

Returns the handle for the current figure. The syntax for its use is

```
handle = gcf
```

where `handle` is the number of the active figure (also its handle).

23.17 GET Get Object Property

23.17.1 Usage

This function allows you to retrieve the value associated with a property. The syntax for its use is

```
value = get(handle,property)
```

where `property` is a string containing the name of the property, and `value` is the value for that property. The type of the variable `value` depends on the property being set. See the help for the properties to see what values you can set.

23.18 GRAY Gray Colormap

23.18.1 Usage

Returns a gray colormap. The syntax for its use is

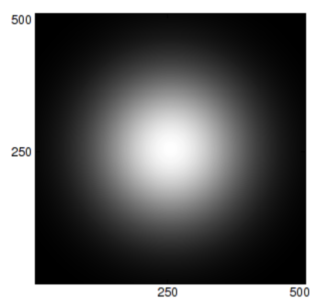
```
y = gray
```

23.18.2 Example

Here is an example of an image displayed with the `gray` colormap

```
--> x = linspace(-1,1,512)'*ones(1,512);
--> y = x';
--> Z = exp(-(x.^2+y.^2)/0.3);
--> image(Z);
--> colormap(gray);
```

which results in the following image



23.19 GRID Plot Grid Toggle Function

23.19.1 Usage

Toggles the drawing of grid lines on the currently active plot. The general syntax for its use is

```
grid(state)
```

where `state` is either

```
grid('on')
```

to activate the grid lines, or

```
grid('off')
```

to deactivate the grid lines. If you specify no argument then **grid** toggles the state of the grid:

```
grid
```

You can also specify a particular axis to the grid command

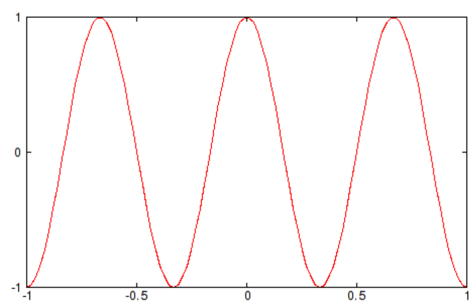
```
grid(handle,...)
```

where **handle** is the handle for a particular axis.

23.19.2 Example

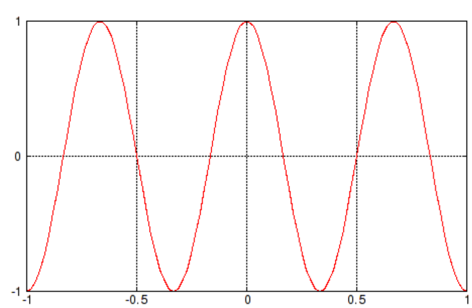
Here is a simple plot without grid lines.

```
--> x = linspace(-1,1);
--> y = cos(3*pi*x);
--> plot(x,y,'r-');
```



Next, we activate the grid lines.

```
--> plot(x,y,'r-');
--> grid on
```



23.20 HIMAGE Create a image object

23.20.1 Usage

Creates a image object and parents it to the current axis. The syntax for its use is

```
handle = himage(property,value,property,value,...)
```

where `property` and `value` are set. The handle ID for the resulting object is returned. It is automatically added to the children of the current axis.

23.21 HLINE Create a line object

23.21.1 Usage

Creates a line object and parents it to the current axis. The syntax for its use is

```
handle = hline(property,value,property,value,...)
```

where `property` and `value` are set. The handle ID for the resulting object is returned. It is automatically added to the children of the current axis.

23.22 HOLD Plot Hold Toggle Function

23.22.1 Usage

Toggles the hold state on the currently active plot. The general syntax for its use is

```
hold(state)
```

where `state` is either

```
hold('on')
```

to turn hold on, or

```
hold('off')
```

to turn hold off. If you specify no argument then `hold` toggles the state of the hold:

```
hold
```

You can also specify a particular axis to the hold command

```
hold(handle,...)
```

where `handle` is the handle for a particular axis.

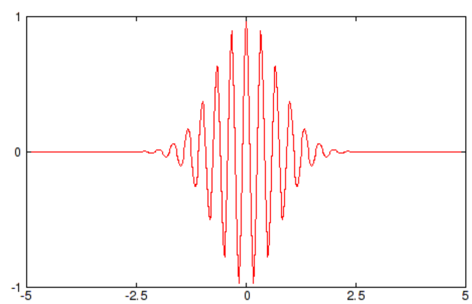
23.22.2 Function Internals

The `hold` function allows one to construct a plot sequence incrementally, instead of issuing all of the plots simultaneously using the `plot` command.

23.22.3 Example

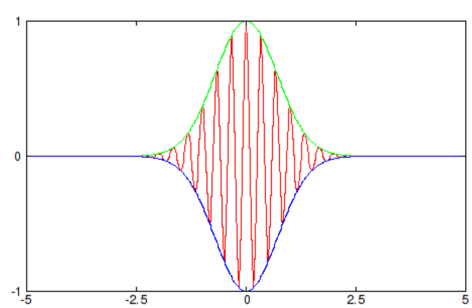
Here is an example of using both the `hold` command and the multiple-argument `plot` command to construct a plot composed of three sets of data. The first is a plot of a modulated Gaussian.

```
--> x = linspace(-5,5,500);
--> t = exp(-x.^2);
--> y = t.*cos(2*pi*x*3);
--> plot(x,y);
```



We now turn the hold state to `'on'`, and add another plot sequence, this time composed of the top and bottom envelopes of the modulated Gaussian. We add the two envelopes simultaneously using a single `plot` command. The fact that `hold` is `'on'` means that these two envelopes are added to (instead of replace) the current contents of the plot.

```
--> plot(x,y);
--> hold on
--> plot(x,t,'g-',x,-t,'b-')
```



23.23 HPOINT Get Point From Window

23.23.1 Usage

This function waits for the user to click on the current figure window, and then returns the coordinates of that click. The generic syntax for its use is

```
[x,y] = hpoint
```

23.24 HSURFACE Create a surface object

23.24.1 Usage

Creates a surface object and parents it to the current axis. The syntax for its use is

```
handle = hsurface(property,value,property,value,...)
```

where `property` and `value` are set. The handle ID for the resulting object is returned. It is automatically added to the children of the current axis.

23.25 HTEXT Create a text object

23.25.1 Usage

Creates a text object and parents it to the current axis. The syntax for its use is

```
handle = htext(property,value,property,value,...)
```

where `property` and `value` are set. The handle ID for the resulting object is returned. It is automatically added to the children of the current axis.

23.26 IMAGE Image Display Function

23.26.1 Usage

The `image` command has the following general syntax

```
handle = image(x,y,C,properties...)
```

where `x` is a two vector containing the `x` coordinates of the first and last pixels along a column, and `y` is a two vector containing the `y` coordinates of the first and last pixels along a row. The matrix `C` constitutes the image data. It must either be a scalar matrix, in which case the image is colormapped using the `colormap` for the current figure. If the matrix is `M x N x 3`, then `C` is intepreted as RGB data, and the image is not colormapped. The `properties` argument is a set of `property/value` pairs that affect the final image. You can also omit the `x` and `y`,

```
handle = image(C, properties...)
```

in which case they default to `x = [1,size(C,2)]` and `y = [1,size(C,1)]`. Finally, you can use the `image` function with only formal arguments

```
handle = image(properties...)
```

To support legacy FreeMat code, you can also use the following form of `image`

```
image(C, zoomfactor)
```

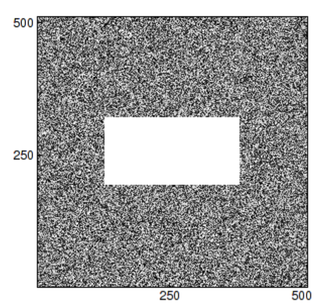
which is equivalent to `image(C)` with the axes removed so that the image takes up the full figure window, and the size of the figure window adjusted to achieve the desired zoom factor using the `zoom` command.

23.26.2 Example

In this example, we create an image that is `512 x 512` pixels square, and set the background to a noise pattern. We set the central `128 x 256` pixels to be white.

```
--> x = rand(512);
--> x((-64:63)+256,(-128:127)+256) = 1.0;
--> figure
ans =
  <int32>  - size: [1 1]
 1
--> image(x)
ans =
  <uint32>  - size: [1 1]
 100002
--> colormap(gray)
```

The resulting image looks like:

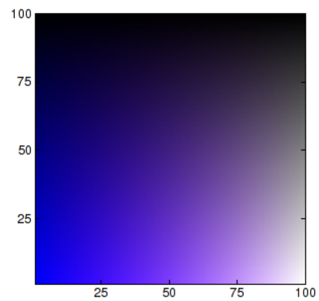


Here is an example of an RGB image

```
--> t = linspace(0,1);
--> red = t'*t;
```

```
--> green = t'*(t.^2);
--> blue = t'*(0*t+1);
--> A(:,:,1) = red;
--> A(:,:,2) = green;
--> A(:,:,3) = blue;
--> image(A);
```

The resulting image looks like:



23.27 IMAGEPROPERTIES Image Object Properties

23.27.1 Usage

Below is a summary of the properties for the axis.

- `alphadata` - `vector` - This is a vector that should contain as many elements as the image data itself `cdata`, or a single scalar. For a single scalar, all values of the image take on the same transparency. Otherwise, the transparency of each pixel is determined by the corresponding value from the `alphadata` vector.

- `alphadatamapping` - `{'scaled','direct','none'}` - For `none` mode (the default), no transparency is applied to the data. For `direct` mode, the vector `alphadata` contains values between @[0,M-1]— where `M` is the length of the alpha map stored in the figure. For `scaled` mode, the `alim` vector for the figure is used to linearly rescale the alpha data prior to lookup in the alpha map.

- `cdata` - `array` - This is either a `M x N` array or an `M x N x 3` array. If the data is `M x N` the image is a scalar image (indexed mode), where the color associated with each image pixel is computed using the colormap and the `cdatamapping` mode. If the data is `M x N x 3` the image is assumed to be in RGB mode, and the colorpanes are taken directly from `cdata` (the colormap is ignored). Note that in this case, the data values must be between @[0,1]— for each color channel and each pixel.

- `cdatamapping` - `{'scaled','direct'}` - For `scaled` (the default), the pixel values are scaled using the `clim` vector for the figure prior to looking up in the colormap. For `direct` mode, the pixel values must be in the range `[0,N-1` where `N` is the number of colors in the colormap.
- `children` - Not used.
- `parent` - `handle` - The axis containing the image.
- `tag` - `string` - You can set this to any string you want.
- `type` - `string` - Set to the string `'image'`.
- `xdata` - `two vector` - contains the x coordinates of the first and last column (respectively). Defaults to `[1,C]` where `C` is the number of columns in the image.
- `ydata` - `two vector` - contains the y coordinates of the first and last row (respectively). Defaults to `[1,R]` where `R` is the number of rows in the image.
- `userdata` - `array` - Available to store any variable you want in the handle object.
- `visible` - `{'on','off'}` - Controls whether the image is visible or not.

23.28 ISHOLD Test Hold Status

23.28.1 Usage

Returns the state of the `hold` flag on the currently active plot. The general syntax for its use is

```
ishold
```

and it returns a logical 1 if `hold` is `on`, and a logical 0 otherwise.

23.29 LEGEND Add Legent to Plot

23.29.1 Usage

This command adds a legend to the current plot. Currently, the following forms of the `legend` command are supported. The first form creates a legend with the given labels for the data series:

```
legend('label1','label2',...)
```

where `'label1'` is the text label associated with data plot 1 and so on. You can also use the `legend` command to control the appearance of the legend in the current plot. To remove the legend from the current plot, use

```
legend('off')
```

To hide the legend for the current plot (but do not remove it)

```
legend('hide')
```

And to show the legend that has been hidden, use

```
legend('show')
```

You can also toggle the display of the box surrounding the legend. Use

```
legend('boxoff')
```

or

```
legend('boxon')
```

to turn the legend box off or on, respectively. To toggle the visible state of the current legend, use

```
legend('toggle')
```

Specifying no arguments at all (apart from an optional location argument as specified below) results in the legend being rebuilt. This form is useful for picking up font changes or relocating the legend.

```
legend
```

By default, the `legend` command places the new legend in the upper right corner of the current plot. To change this behavior, use the `'location'` specifier (must be the last two options to the command)

```
legend(...,'location',option)
```

where `option` takes on the following possible values

- `north`,`N` - top center of plot
- `south`,`S` - bottom center of plot
- `east`,`E` - middle right of plot
- `west`,`W` - middle left of plot
- `northeast`,`NE` - top right of plot (default behavior)
- `northwest`,`NW` - top left of plot
- `southeast`,`SE` - bottom right of plot
- `southwest`,`SW` - bottom left of plot

This implementation of `legend` is incomplete relative to the MATLAB API. The functionality will be improved in future versions of FreeMat.

23.30 LINEPROPERTIES Line Series Object Properties

23.30.1 Usage

Below is a summary of the properties for a line series.

- `color` - `colorspec` - The color that is used to draw the line.
- `children` - Not used.
- `displayname` - The name of this line series as it appears in a legend.
- `linestyle` - `{'-','--',':','-.','none'}` - The style of the line.
- `linewidth` - `scalar` - The width of the line.
- `marker` - `{'+','o','*','.','x','square','s','diamond','d','^','v','>','<'}` - The marker for data points on the line. Some of these are redundant, as `'square'` `'s'` are synonyms, and `'diamond'` and `'d'` are also synonyms.
- `markeredgecolor` - `colorspec` - The color used to draw the marker. For some of the markers (circle, square, etc.) there are two colors used to draw the marker. This property controls the edge color (which for unfilled markers) is the primary color of the marker.
- `markerfacecolor` - `colorspec` - The color used to fill the marker. For some of the markers (circle, square, etc.) there are two colors used to fill the marker.
- `markersize` - `scalar` - Control the size of the marker. Defaults to 6, which is effectively the radius (in pixels) of the markers.
- `parent` - `handle` - The axis that contains this object.
- `tag` - `string` - A string that can be used to tag the object.
- `type` - `string` - Returns the string `'line'`.
- `visible` - `{'on','off'}` - Controls visibility of the the line.
- `xdata` - `vector` - Vector of x coordinates of points on the line. Must be the same size as the `ydata` and `zdata` vectors.
- `ydata` - `vector` - Vector of y coordinates of points on the line. Must be the same size as the `xdata` and `zdata` vectors.
- `zdata` - `vector` - Vector of z coordinates of points on the line. Must be the same size as the `xdata` and `ydata` vectors.
- `xdatamode` - `{'auto','manual'}` - When set to `'auto'` FreeMat will autogenerate the x coordinates for the points on the line. These values will be `1,..,N` where `N` is the number of points in the line.
- `userdata` - `array` - Available to store any variable you want in the handle object.

23.31 LOGLOG Log-Log Plot Function

23.31.1 Usage

This command has the exact same syntax as the `plot` command:

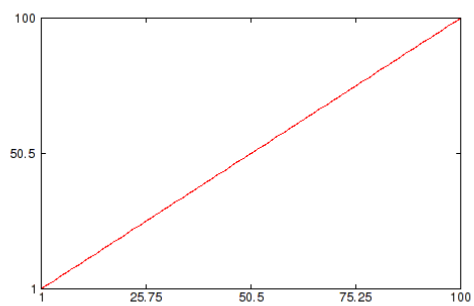
```
loglog(<data 1>,{linespec 1},<data 2>,{linespec 2}...,properties...)
```

in fact, it is a simple wrapper around `plot` that sets the x and y axis to have a logarithmic scale.

23.31.2 Example

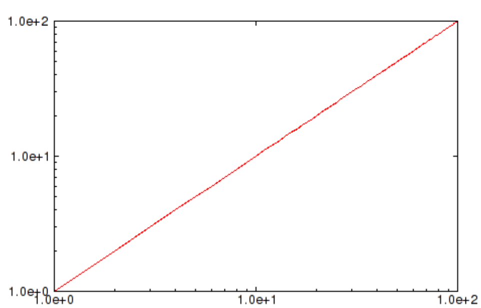
Here is an example of a doubly exponential signal plotted first on a linear plot:

```
--> x = linspace(1,100);
--> y = x;
--> plot(x,y,'r-');
```



and now on a log-log plot

```
--> loglog(x,y,'r-');
```



23.32 NEWPLOT Get Handle For Next Plot

23.32.1 Usage

Returns the handle for the next plot operation. The general syntax for its use is

```
h = newplot
```

This routine checks the `nextplot` properties of the current figure and axes to see if they are set to `replace` or not. If the figures `nextplot` property is set to replace, the current figure is cleared. If the axes `nextplot` property is set to `replace` then the axes are cleared for the next operation.

23.33 PLOT Plot Function

23.33.1 Usage

This is the basic plot command for FreeMat. The general syntax for its use is

```
plot(<data 1>,{linespec 1},<data 2>,{linespec 2}...,properties...)
```

where the `<data>` arguments can have various forms, and the `linespec` arguments are optional. We start with the `<data>` term, which can take on one of multiple forms:

- *Vector Matrix Case* – In this case the argument data is a pair of variables. A set of `x` coordinates in a numeric vector, and a set of `y` coordinates in the columns of the second, numeric matrix. `x` must have as many elements as `y` has columns (unless `y` is a vector, in which case only the number of elements must match). Each column of `y` is plotted sequentially against the common vector `x`.

- *Unpaired Matrix Case* – In this case the argument data is a single numeric matrix `y` that constitutes the `y`-values of the plot. An `x` vector is synthesized as `x = 1:length(y)`, and each column of `y` is plotted sequentially against this common `x` axis.

- *Complex Matrix Case* – Here the argument data is a complex matrix, in which case, the real part of each column is plotted against the imaginary part of each column. All columns receive the same line styles.

Multiple data arguments in a single plot command are treated as a *sequence*, meaning that all of the plots are overlapped on the same set of axes. The `linespec` is a string used to change the characteristics of the line. In general, the `linespec` is composed of three optional parts, the `colorspec`, the `symbolspec` and the `linestylespec` in any order. Each of these specifications is a single character that determines the corresponding characteristic. First, the `colorspec`:

- `'r'` - Color Red
- `'g'` - Color Green
- `'b'` - Color Blue
- `'k'` - Color Black

- `'c'` - Color Cyan
- `'m'` - Color Magenta
- `'y'` - Color Yellow

The `symbolspec` specifies the (optional) symbol to be drawn at each data point:

- `'.'` - Dot symbol
- `'o'` - Circle symbol
- `'x'` - Times symbol
- `'+'` - Plus symbol
- `'*'` - Asterisk symbol
- `'s'` - Square symbol
- `'d'` - Diamond symbol
- `'v'` - Downward-pointing triangle symbol
- `'^'` - Upward-pointing triangle symbol
- `'<'` - Left-pointing triangle symbol
- `'>'` - Right-pointing triangle symbol

The `linestylespec` specifies the (optional) line style to use for each data series:

- `'-'` - Solid line style
- `':'` - Dotted line style
- `';'` - Dot-Dash-Dot-Dash line style
- '—'— - Dashed line style

For sequences of plots, the `linespec` is recycled with color order determined by the properties of the current axes. You can also use the `properties` argument to specify handle properties that will be inherited by all of the plots generated during this event. Finally, you can also specify the handle for the axes that are the target of the `plot` operation.

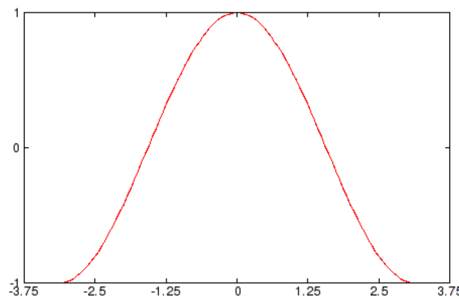
```
handle = plot(handle,...)
```

23.33.2 Example

The most common use of the `plot` command probably involves the vector-matrix paired case. Here, we generate a simple cosine, and plot it using a red line, with no symbols (i.e., a `linespec` of `'r-'`).

```
--> x = linspace(-pi,pi);
--> y = cos(x);
--> plot(x,y,'r-');
```

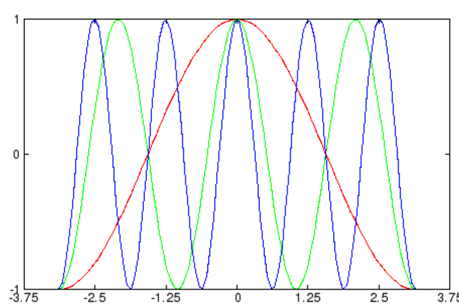
which results in the following plot.



Next, we plot multiple sinusoids (at different frequencies). First, we construct a matrix, in which each column corresponds to a different sinusoid, and then plot them all at once.

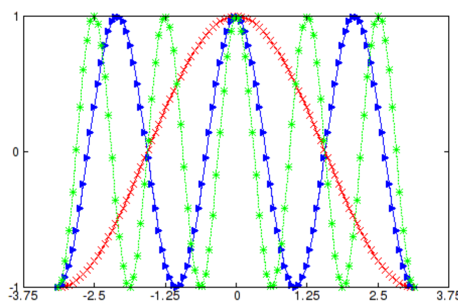
```
--> x = linspace(-pi,pi);
--> y = [cos(x(:)),cos(3*x(:)),cos(5*x(:))];
--> plot(x,y);
```

In this case, we do not specify a `linespec`, so that we cycle through the colors automatically (in the order listed in the previous section).



This time, we produce the same plot, but as we want to assign individual `linespecs` to each line, we use a sequence of arguments in a single plot command, which has the effect of plotting all of the data sets on a common axis, but which allows us to control the `linespec` of each plot. In the following example, the first line (harmonic) has red, solid lines with times symbols marking the data points, the second line (third harmonic) has blue, solid lines with right-pointing triangle symbols, and the third line (fifth harmonic) has green, dotted lines with asterisk symbols.

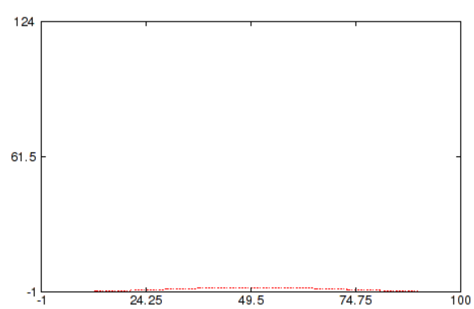
```
--> plot(x,y(:,1),'rx-',x,y(:,2),'b>-',x,y(:,3),'g*:');
```



The second most frequently used case is the unpaired matrix case. Here, we need to provide only one data component, which will be automatically plotted against a vector of natural number of the appropriate length. Here, we use a plot sequence to change the style of each line to be dotted, dot-dashed, and dashed.

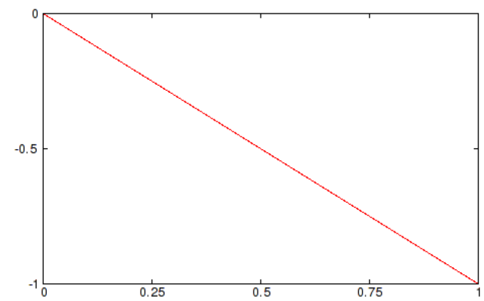
```
--> plot(y(:,1),'r:',y(:,2),'b;',y(:,3),'g|');
```

Note in the resulting plot that the `x`-axis no longer runs from `[-pi,pi]`, but instead runs from `[1,100]`.



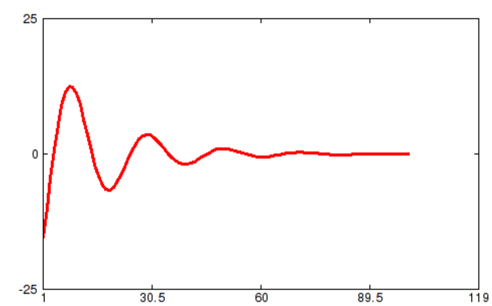
The final case is for complex matrices. For complex arguments, the real part is plotted against the imaginary part. Hence, we can generate a 2-dimensional plot from a vector as follows.

```
--> y = cos(2*x) + i * cos(3*x);
--> plot(y);
```



Here is an example of using the handle properties to influence the behavior of the generated lines.

```
--> t = linspace(-3,3);
--> plot(cos(5*t).*exp(-t),'r-','linewidth',3);
```



23.34 PLOT3 Plot 3D Function

23.34.1 Usage

This is the 3D plot command. The general syntax for its use is

```
plot3(X,Y,Z,{linespec 1},X,Y,Z,{linespec 2},...,properties...)
```

where `X` `Y` and `Z` are the coordinates of the points on the 3D line. Note that in general, all three should be vectors. If some or all of the quantities are matrices, then FreeMat will attempt to expand the vector arguments to the same size, and then generate multiple plots, one for each column of the matrices. The linespec is optional, see `plot` for details. You can specify `properties` for the generated line plots. You can also specify a handle as an axes to target

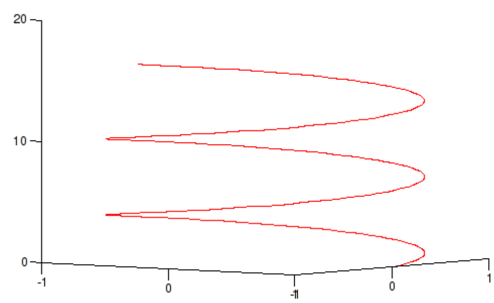
```
plot3(handle,...)
```

23.34.2 Example

Here is a simple example of a 3D helix.

```
--> t = linspace(0,5*pi,200);
--> x = cos(t); y = sin(t); z = t;
--> plot3(x,y,z);
--> view(3);
```

Shown here



23.35 POINT Get Axis Position From Mouse Click

23.35.1 Usage

Returns information about the currently displayed image based on a use supplied mouse-click. The general syntax for its use is

```
t = point
```

The returned vector `y` has two elements:

$$t = [x, y]$$

where `x,y` are the coordinates in the current axes of the click. This function has changed since FreeMat 1.10. If the click is not inside the active area of any set of axes, a pair of NaNs are returned.

23.36 PRINT Print a Figure To A File

23.36.1 Usage

This function “prints” the currently active fig to a file. The generic syntax for its use is

```
print(filename)
```

or, alternately,

```
print filename
```

where `filename` is the (string) filename of the destined file. The current fig is then saved to the output file using a format that is determined by the extension of the filename. The exact output formats may vary on different platforms, but generally speaking, the following extensions should be supported cross-platform:

- `jpg`, `jpeg` – JPEG file
- `pdf` – Portable Document Format file
- `png` – Portable Net Graphics file

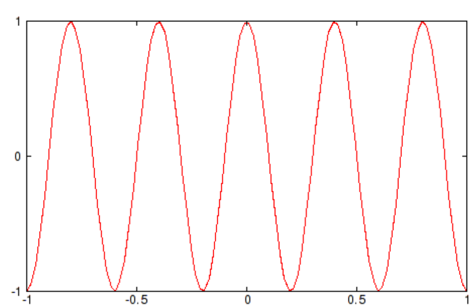
Postscript (PS, EPS) is supported on non-Mac-OSX Unix only. Note that only the fig is printed, not the window displaying the fig. If you want something like that (essentially a window-capture) use a seperate utility or your operating system’s built in screen capture ability.

23.36.2 Example

Here is a simple example of how the figures in this manual are generated.

```
--> x = linspace(-1,1);
--> y = cos(5*pi*x);
--> plot(x,y,'r-');
--> print('printfig1.jpg')
--> print('printfig1.png')
```

which creates two plots `printfig1.png`, which is a Portable Net Graphics file, and `printfig1.jpg` which is a JPEG file.



23.37 PVALID Validate Property Name

23.37.1 Usage

This function checks to see if the given string is a valid property name for an object of the given type. The syntax for its use is

```
  b = pvalid(type,propertyname)
```

where `string` is a string that contains the name of a valid graphics object type, and `propertyname` is a string that contains the name of the property to test for.

23.37.2 Example

Here we test for some properties on an `axes` object.

```
--> pvalid('axes','type')
ans =
  <logical>  - size: [1 1]
 1
--> pvalid('axes','children')
ans =
  <logical>  - size: [1 1]
 1
--> pvalid('axes','foobar')
ans =
  <logical>  - size: [1 1]
 0
```

23.38 SEMILOGX Semilog X Axis Plot Function

23.38.1 Usage

This command has the exact same syntax as the `plot` command:

```
  semilogx(<data 1>,{linespec 1},<data 2>,{linespec 2}...,properties...)
```

in fact, it is a simple wrapper around `plot` that sets the x axis to have a logarithmic scale.

23.38.2 Example

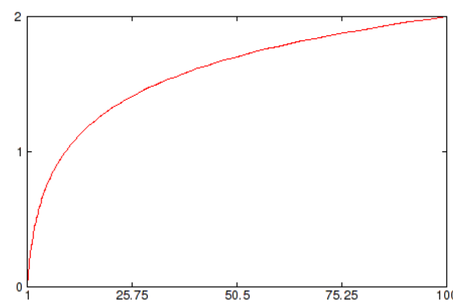
Here is an example of an exponential signal plotted first on a linear plot:

```
--> y = linspace(0,2);
--> x = (10).^y
x =
  <double>  - size: [1 100]
```

```
Columns 1 to 3
   1.0000000000000000    1.0476157527896648    1.0974987654930561

Columns 4 to 6
   1.1497569953977358    1.2045035402587823    1.2618568830660204

Columns 7 to 9
   1.3219411484660291    1.3848863713938731    1.4508287784959397

Columns 10 to 12
   1.5199110829529336    1.5922827933410924    1.6681005372000588

Columns 13 to 15
   1.7475284000076838    1.8307382802953682    1.9179102616724886

Columns 16 to 18
   2.0092330025650473    2.1049041445120205    2.2051307399030455

Columns 19 to 21
   2.3101297000831598    2.4201282647943820    2.5353644939701119

Columns 22 to 24
   2.6560877829466865    2.7825594022071245    2.9150530628251765

Columns 25 to 27
   3.0538555088334154    3.1992671377973836    3.3516026509388426

Columns 28 to 30
   3.5111917342151311    3.6783797718286335    3.8535285937105295

Columns 31 to 33
   4.0370172585965545    4.2292428743894988    4.4306214575838814

Columns 34 to 36
   4.6415888336127784    4.8626015800653537    5.0941380148163793

Columns 37 to 39
   5.3366992312063095    5.5908101825122243    5.8570208180566654

Columns 40 to 42
   6.1359072734131725    6.4280731172843204    6.7341506577508214

Columns 43 to 45
   7.0548023107186433    7.3907220335257788    7.7426368268112693

Columns 46 to 48
```

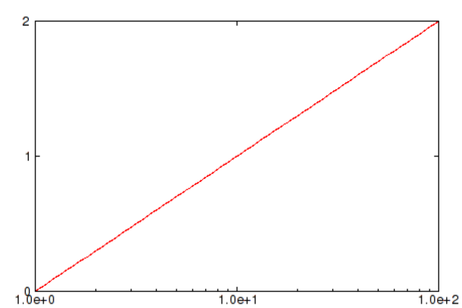
```
   8.1113083078968700    8.4975343590864423    8.9021508544503867

Columns 49 to 51
   9.3260334688321986    9.7700995729922546   10.2353102189902625

Columns 52 to 54
  10.7226722201032310   11.2332403297802763   11.7681195243499843

Columns 55 to 57
  12.3284673944206595   12.9154966501488406   13.5304777457980681

Columns 58 to 60
  14.1747416292680555   14.8496826225446501   15.5567614393047151

Columns 61 to 63
  16.2975083462064418   17.0735264747069060   17.8864952905743522

Columns 64 to 66
  18.7381742286038389   19.6304065004027137   20.5651230834865153

Columns 67 to 69
  21.5443469003188319   22.5701971963392047   23.6448941264540728

Columns 70 to 72
  24.7707635599171141   25.9502421139973585   27.1858824273293997

Columns 73 to 75
  28.4803586843580199   29.8364724028333868   31.2571584968823686

Columns 76 to 78
  32.7454916287772804   34.3046928631491710   35.9381366380462737

Columns 79 to 81
  37.6493580679246733   39.4420605943765636   41.3201240011533670

Columns 82 to 84
  43.2876128108305949   45.3487850812858184   47.5081016210279543

Columns 85 to 87
  49.7702356433211150   52.1400828799968465   54.6227721768434265

Columns 88 to 90
  57.2236765935021694   59.9484250318940894   62.8029144183425316

Columns 91 to 93
  65.7933224657567877   68.9261210434969911   72.2080901838546367
```

```
Columns 94 to 96
  75.6463327554629075   79.2482898353917307   83.0217568131974417

Columns 97 to 99
  86.9749002617783447   91.1162756115489145   95.4548456661834166

Columns 100 to 100
 100.0000000000000000
--> plot(x,y,'r-');
```



and now with a logarithmic x axis

```
--> semilogx(x,y,'r-');
```



23.39 SEMILOGY Semilog Y Axis Plot Function

23.39.1 Usage

This command has the exact same syntax as the `plot` command:

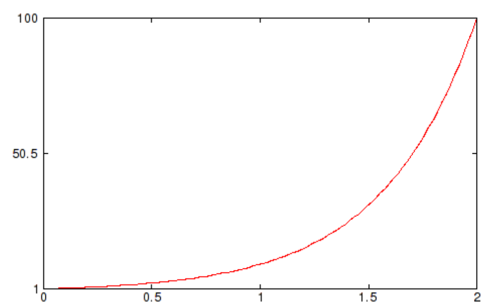
```
semilogy(<data 1>,{linespec 1},<data 2>,{linespec 2}...,properties...)
```

in fact, it is a simple wrapper around `plot` that sets the y axis to have a logarithmic scale.

23.39.2 Example

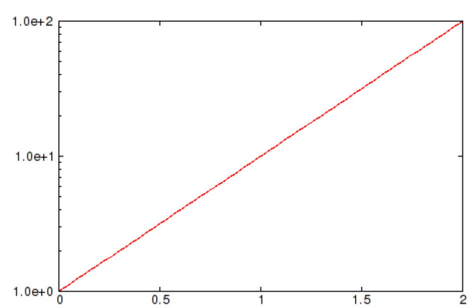
Here is an example of an exponential signal plotted first on a linear plot:

```
--> x = linspace(0,2);
--> y = 10.0.^x;
--> plot(x,y,'r-');
```



and now with a logarithmic y axis

```
--> semilogy(x,y,'r-');
```



23.40 SET Set Object Property

23.40.1 Usage

This function allows you to change the value associated with a property. The syntax for its use is

```
set(handle,property,value,property,value,...)
```

where `property` is a string containing the name of the property, and `value` is the value for that property. The type of the variable `value` depends on the property being set. See the help for the properties to see what values you can set.

23.41 SIZEFIG Set Size of an Fig Window

23.41.1 Usage

The `sizefig` function changes the size of the currently selected fig window. The general syntax for its use is

```
sizefig(width,height)
```

where `width` and `height` are the dimensions of the fig window.

23.42 SUBPLOT Subplot Function

23.42.1 Usage

This function divides the current figure into a 2-dimensional grid, each of which can contain a plot of some kind. The function has a number of syntaxes. The first version

```
subplot(row,col,num)
```

which either activates subplot number `num`, or sets up a subplot grid of size `row x col`, and then activates `num`. You can also set up subplots that cover multiple grid elements

```
subplot(row,col,[vec])
```

where `vec` is a set of indexes covered by the new subplot. Finally, as a shortcut, you can specify a string with three components

```
subplot('mnp')
```

or using the alternate notation

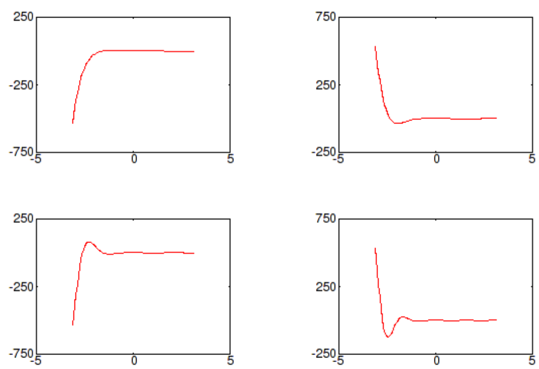
```
subplot mnp
```

where `m` is the number of rows, `n` is the number of columns and `p` is the index.

23.42.2 Example

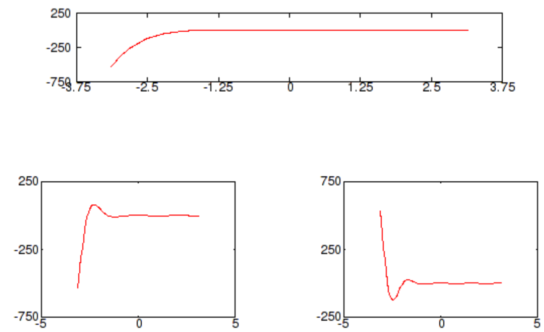
Here is the use of `subplot` to set up a `2 x 2` grid of plots

```
--> t = linspace(-pi,pi);
--> subplot(2,2,1)
ans =
  <uint32>  - size: [1 1]
 100001
--> plot(t,cos(t).*exp(-2*t));
--> subplot(2,2,2);
--> plot(t,cos(t*2).*exp(-2*t));
--> subplot(2,2,3);
--> plot(t,cos(t*3).*exp(-2*t));
--> subplot(2,2,4);
--> plot(t,cos(t*4).*exp(-2*t));
```



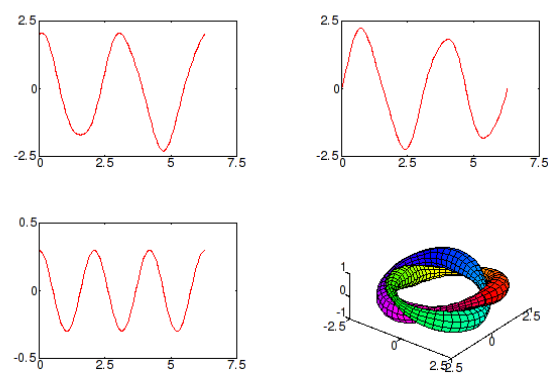
Here we use the second form of `subplot` to generate one subplot that is twice as large.

```
--> t = linspace(-pi,pi);
--> subplot(2,2,[1,2])
ans =
  <uint32>  - size: [1 1]
 100001
--> plot(t,cos(t).*exp(-2*t));
--> subplot(2,2,3);
--> plot(t,cos(t*3).*exp(-2*t));
--> subplot(2,2,4);
--> plot(t,cos(t*4).*exp(-2*t));
```



Note that the subplots can contain any handle graphics objects, not only simple plots.

```
--> t=0:(2*pi/100):(2*pi);
--> x=cos(t*2).*(2+sin(t*3)*.3);
--> y=sin(t*2).*(2+sin(t*3)*.3);
--> z=cos(t*3)*.3;
--> subplot(2,2,1)
ans =
  <uint32>  - size: [1 1]
 100001
--> plot(t,x);
--> subplot(2,2,2);
--> plot(t,y);
--> subplot(2,2,3);
--> plot(t,z);
--> subplot(2,2,4);
--> tubeplot(x,y,z,0.14*sin(t*5)+.29,t,10)
--> axis equal
--> view(3)
```



23.43 SURF Surface Plot Function

23.43.1 Usage

This routine is used to create a surface plot of data. A surface plot is a 3D surface defined by the xyz coordinates of its vertices and optionally by the color at the vertices. The most general syntax for the `surf` function is

```
  h = surf(X,Y,Z,C,properties...)
```

Where `X` is a matrix or vector of `x` coordinates, `Y` is a matrix or vector of `y` coordinates, `Z` is a 2D matrix of coordinates, and `C` is a 2D matrix of color values (the colormap for the current fig is applied). In general, `X` and `Y` should be the same size as `Z`, but FreeMat will expand vectors to match the matrix if possible. If you want the color of the surface to be defined by the height of the surface, you can omit `C`

```
  h = surf(X,Y,Z,properties...)
```

in which case `C=Z`. You can also eliminate the `X` and `Y` matrices in the specification

```
  h = surf(Z,properties)
```

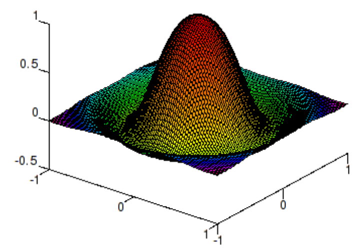
in which case they are set to `1:size(Z,2)` and `1:size(Y,2)` respectively. You can also specify a handle as the target of the `surf` command via

```
  h = surf(handle,...)
```

23.43.2 Example

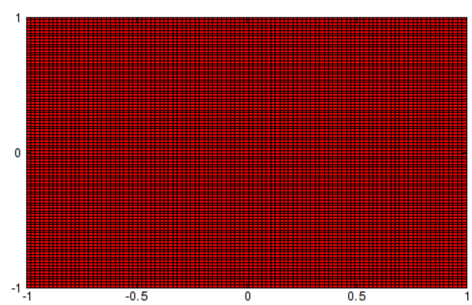
Here we generate a surface specifying all four components.

```
--> x = repmat(linspace(-1,1),[100,1]);
--> y = x';
--> r = x.^2+y.^2;
--> z = exp(-r*3).*cos(5*r);
--> c = r;
--> surf(x,y,z,c)
ans =
  <uint32>  - size: [1 1]
 100002
--> axis equal
--> view(3)
```



If we allow FreeMat to specify the color component, we see that the colorfield is the same as the height

```
--> surf(x,y,z)
ans =
  <uint32>  - size: [1 1]
 100002
```



23.44 SURFACEPROPERTIES Surface Object Properties

23.44.1 Usage

Below is a summary of the properties for the axis.

- `alphadata` - `vector` - This is a vector that should contain as many elements as the surface data itself `cdata`, or a single scalar. For a single scalar, all values of the surface take on the same transparency. Otherwise, the transparency of each pixel is determined by the corresponding value from the `alphadata` vector.

- `alphadatamapping` - `{'scaled','direct','none'}` - For `none` mode (the default), no transparency is applied to the data. For `direct` mode, the vector `alphadata` contains values between @[0,M-1]— where `M` is the length of the alpha map stored in the figure. For `scaled` mode, the `alim` vector for the figure is used to linearly rescale the alpha data prior to lookup in the alpha map.

- `ambientstrength` - Not used.

- `backfacelighting` - Not used.

- `cdata` - `array` - This is either a `M x N` array or an `M x N x 3` array. If the data is `M x N` the surface is a scalar surface (indexed mode), where the color associated with each surface pixel is computed using the colormap and the `cdatamapping` mode. If the data is `M x N x 3` the surface is assumed to be in RGB mode, and the colorpanes are taken directly from `cdata` (the colormap is ignored). Note that in this case, the data values must be between @[0,1]— for each color channel and each point on the surface.

- `cdatamapping` - `{'scaled','direct'}` - For `scaled` (the default), the pixel values are scaled using the `clim` vector for the figure prior to looking up in the colormap. For `direct` mode, the pixel values must be in the range `[0,N-1` where `N` is the number of colors in the colormap.

- `children` - Not used.

- `diffusestrength` - Not used.

- `edgealpha` - `{'flat','interp','scalar'}` - Controls how the transparency is mapped for the edges of the surface.

- `edgecolor` - `{'flat','interp','none',colorspec}` - Specifies how the edges are colored. For `'flat'` the edges are flat colored, meaning that the line segments that make up the edges are not shaded. The color for the line is determined by the first edge point it is connected to.

- `edgelighting` - Not used.

- `facealpha` - `{'flat','interp','texturemap',scalar}` - Controls how the transparency of the faces of the surface are controlled. For flat shading, the faces are constant transparency. For interp mode, the faces are smoothly transparently mapped. If set to a scalar, all faces have the same transparency.

- `facecolor` - `{'none','flat','interp',colorspec}` - Controls how the faces are colored. For `'none'` the faces are uncolored, and the surface appears as a mesh without hidden lines removed. For `'flat'` the surface faces have a constant color. For `'interp'` smooth shading is applied to the surface. And if a colorspec is provided, then the faces all have the same color.

- `facelighting` - Not used.

- `linestyle` - `{'-','--',':','-.','none'}` - The style of the line used to draw the edges.

- `linewidth` - `scalar` - The width of the line used to draw the edges.

- marker - {'+','o','*','.','x','square','s','diamond','d','^','v','>','<'} - The marker for data points on the line. Some of these are redundant, as 'square' 's' are synonyms, and 'diamond' and 'd' are also synonyms.
- markeredgecolor - colorspec - The color used to draw the marker. For some of the markers (circle, square, etc.) there are two colors used to draw the marker. This property controls the edge color (which for unfilled markers) is the primary color of the marker.
- markerfacecolor - colorspec - The color used to fill the marker. For some of the markers (circle, square, etc.) there are two colors used to fill the marker.
- markersize - scalar - Control the size of the marker. Defaults to 6, which is effectively the radius (in pixels) of the markers.
- meshstyle - {'both','rows','cols} - This property controls how the mesh is drawn for the surface. For rows and cols modes, only one set of edges is drawn.
- normalmode - Not used.
- parent - handle - The axis containing the surface.
- specularcolorreflectance - Not used.
- specularexponent - Not used.
- specularstrength - Not used.
- tag - string - You can set this to any string you want.
- type - string - Set to the string 'surface'.
- userdata - array - Available to store any variable you want in the handle object.
- vertexnormals - Not used.
- xdata - array - Must be a numeric array of size M x N which contains the x location of each point in the defined surface. Must be the same size as ydata and zdata.
- xdatamode - {'auto','manual'} - When set to auto then FreeMat will automatically generate the x coordinates.
- ydata - array - Must be a numeric array of size M x N which contains the y location of each point in the defined surface. Must be the same size as xdata and zdata.
- ydatamode - {'auto','manual'} - When set to auto then FreeMat will automatically generate the y coordinates.
- zdata - array - Must be a numeric array of size M x N which contains the y location of each point in the defined surface. Must be the same size as xdata and ydata.
- visible - {'on','off'} - Controls whether the surface is visible or not.

23.45 TEXT Add Text Label to Plot

23.45.1 Usage

Adds a text label to the currently active plot. The general syntax for it is use is either

```
text(x,y,'label')
```

where `x` and `y` are both vectors of the same length, in which case the text `'label'` is added to the current plot at each of the coordinates `x(i),y(i)` (using the current axis to map these to screen coordinates). The second form supplies a cell-array of strings as the second argument, and allows you to place many labels simultaneously

```
text(x,y,{'label1','label2',....})
```

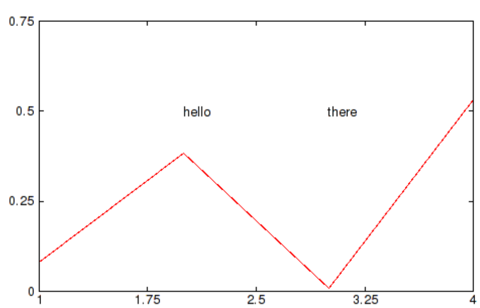
where the number of elements in the cell array must match the size of vectors `x` and `y`. You can also specify properties for the labels via

```
handles = text(x,y,{labels},properties...)
```

23.45.2 Example

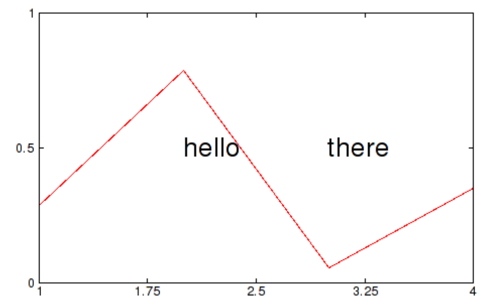
Here is an example of a few labels being added to a random plot:

```
--> plot(rand(1,4))
--> text([2,3],[0.5,0.5],{'hello','there'})
```



Here is the same example, but with larger labels:

```
--> plot(rand(1,4))
--> text([2,3],[0.5,0.5],{'hello','there'},'fontsize',20)
```



23.46 TEXTPROPERTIES Text Object Properties

23.46.1 Usage

Below is a summary of the properties for a text object.

- `boundingbox` - `four vector` - The size of the bounding box containing the text (in pixels). May contain negative values if the text is slanted.
- `children` - Not used.
- `string` - `string` - The text contained in the label.
- `extent` - Not used.
- `horizontalalignment` - `{'left','center','right'}` - Controls the alignment of the text relative to the specified position point.
- `position` - `three vector` - The position of the label in axis coordinates.
- `rotation` - `scalar` - The rotation angle (in degrees) of the label.
- `units` - Not used.
- `verticalalignment` - `{'top','bottom','middle'}` - Controls the alignment fo the text relative to the specified position point in the vertical position.
- `backgroundcolor` - `colorspec` - The color used to fill in the background rectangle for the label. Normally this is `none`.
- `edgecolor` - `colorspec` - The color used to draw the bounding rectangle for the label. Normally this is `none`.
- `linewidth` - `scalar` - The width of the line used to draw the border.
- `linestyle` - `{'-','--',':','-.','none'}` - The style of the line used to draw the border.

- `margin` - `scalar` - The amount of spacing to place around the text as padding when drawing the rectangle.
- `fontangle` - `{'normal','italic','oblique'}` - The angle of the fonts used for the labels.
- `fontsize` - `scalar` - The size of fonts used for the text.
- `fontunits` - Not used.
- `fontweight` - `{'normal','bold','light','demi'}` - The weight of the font used for the label
- `visible` - `{'on','off'}` - Controls visibility of the the line.
- `color` - `colorspec` - The color of the text of the label.
- `children` - Not used.
- `parent` - The handle of the axis that owns this label.
- `tag` - `string` - A string that can be used to tag the object.
- `type` - `string` - Returns the string `'text'`.
- `userdata` - `array` - Available to store any variable you want in the handle object.

23.47 TITLE Plot Title Function

23.47.1 Usage

This command adds a title to the plot. The general syntax for its use is

```
title('label')
```

or in the alternate form

```
title 'label'
```

or simply

```
title label
```

Here `label` is a string variable. You can also specify properties for the label, and a handle to serve as a target for the operation

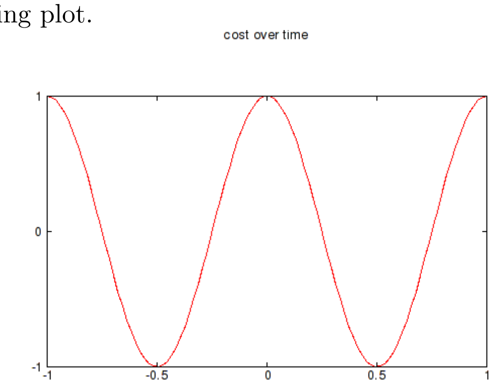
```
title(handle,'label',properties...)
```

23.47.2 Example

Here is an example of a simple plot with a title.

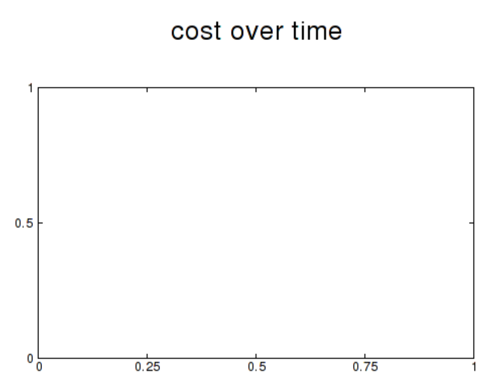
```
--> x = linspace(-1,1);
--> y = cos(2*pi*x);
--> plot(x,y,'r-');
--> title('cost over time');
```

which results in the following plot.



We now increase the size of the font using the properties of the `label`

```
--> title('cost over time','fontsize',20);
```



23.48 TUBEPLOT Creates a Tubeplot

23.48.1 Usage

This `tubeplot` function is from the tubeplot package written by Anders Sandberg. The simplest syntax for the `tubeplot` routine is

```
tubeplot(x,y,z)
```

plots the basic tube with radius 1, where `x,y,z` are vectors that describe the tube. If the radius of the tube is to be varied, use the second form

```
tubeplot(x,y,z,r)
```

which plots the basic tube with variable radius r (either a vector or a scalar value). The third form allows you to specify the coloring using a vector of values:

```
tubeplot(x,y,z,r,v)
```

where the coloring is now dependent on the values in the vector `v`. If you want to create a tube plot with a greater degree of tangential subdivisions (i.e., the tube is more circular, use the form

```
tubeplot(x,y,z,r,v,s)
```

where `s` is the number of tangential subdivisions (default is 6) You can also use `tubeplot` to calculate matrices to feed to `mesh` and `surf`.

```
[X,Y,Z]=tubeplot(x,y,z)
```

returns `N x 3` matrices suitable for mesh or surf.

Note that the tube may pinch at points where the normal and binormal misbehaves. It is suitable for general space curves, not ones that contain straight sections. Normally the tube is calculated using the Frenet frame, making the tube minimally twisted except at inflexion points.

To deal with this problem there is an alternative frame:

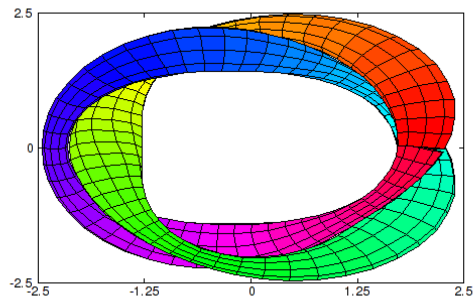
```
tubeplot(x,y,z,r,v,s,vec)
```

calculates the tube by setting the normal to the cross product of the tangent and the vector vec. If it is chosen so that it is always far from the tangent vector the frame will not twist unduly.

23.48.2 Example

Here is an example of a `tubeplot`.

```
--> t=0:(2*pi/100):(2*pi);
--> x=cos(t*2).*(2+sin(t*3)*.3);
--> y=sin(t*2).*(2+sin(t*3)*.3);
--> z=cos(t*3)*.3;
--> tubeplot(x,y,z,0.14*sin(t*5)+.29,t,10);
```



Written by Anders Sandberg, asa@nada.kth.se, 2005

23.49 VIEW Set Graphical View

23.49.1 Usage

The `view` function sets the view into the current plot. The simplest form is

```
view(n)
```

where `n=2` sets a standard view (azimuth 0 and elevation 90), and `n=3` sets a standard 3D view (azimuth 37.5 and elevation 30). With two arguments,

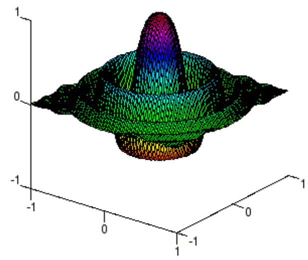
```
view(az,el)
```

you set the viewpoint to azimuth `az` and elevation `el`.

23.49.2 Example

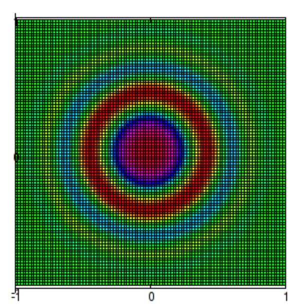
Here is a 3D surface plot shown with a number of viewpoints. First, the default view for a 3D plot.

```
--> x = repmat(linspace(-1,1),[100,1]);
--> y = x';
--> r = x.^2+y.^2;
--> z = exp(-r*3).*cos(5*pi*r);
--> surf(x,y,z);
--> axis equal
--> view(3)
```



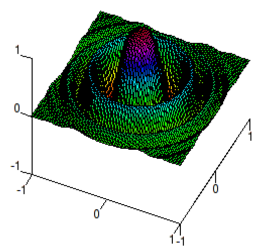
Next, we look at it as a 2D plot

```
--> surf(x,y,z);
--> axis equal
--> view(2)
```



Finally, we generate a different view of the same surface.

```
--> surf(x,y,z);
--> axis equal
--> view(25,50);
```



23.50 WINLEV Image Window-Level Function

23.50.1 Usage

Adjusts the data range used to map the current image to the current colormap. The general syntax for its use is

```
winlev(window,level)
```

where `window` is the new window, and `level` is the new level, or

```
winlev
```

in which case it returns a vector containing the current window and level for the active image.

23.50.2 Function Internals

FreeMat deals with scalar images on the range of `[0,1]`, and must therefor map an arbitrary image `x` to this range before it can be displayed. By default, the `image` command chooses

$$\text{window} = \max x - \min x,$$

and

$$\text{level} = \frac{\text{window}}{2}$$

This ensures that the entire range of image values in `x` are mapped to the screen. With the `winlev` function, you can change the range of values mapped. In general, before display, a pixel `x` is mapped to `[0,1]` via:

$$\max\left(0, \min\left(1, \frac{x - \text{level}}{\text{window}}\right)\right)$$

23.50.3 Examples

The window level function is fairly easy to demonstrate. Consider the following image, which is a Gaussian pulse image that is very narrow:

```
--> t = linspace(-1,1,256);
--> xmat = ones(256,1)*t; ymat = xmat';
--> A = exp(-(xmat.^2 + ymat.^2)*100);
--> image(A);
```

The data range of `A` is `[0,1]`, as we can verify numerically:

```
--> min(A(:))
ans =
  <double>  - size: [1 1]
 1.3838965267367376e-87
--> max(A(:))
ans =
  <double>  - size: [1 1]
 0.9969289851428387
```

To see the tail behavior, we use the `winlev` command to force FreeMat to map a smaller range of `A` to the colormap.

```
--> image(A);
--> winlev(1e-4,0.5e-4)
```

The result is a look at more of the tail behavior of `A`. We can also use the winlev function to find out what the window and level are once set, as in the following example.

```
--> image(A);
--> winlev(1e-4,0.5e-4)
--> winlev
ans =
  <double>  - size: [1 1]
 0.0001
```

23.51 XLABEL Plot X-axis Label Function

23.51.1 Usage

This command adds a label to the x-axis of the plot. The general syntax for its use is

```
  xlabel('label')
```

or in the alternate form

```
  xlabel 'label'
```

or simply

```
xlabel label
```

Here `label` is a string variable. You can also specify properties for that label using the syntax

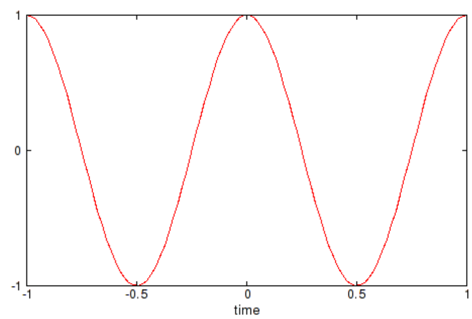
```
xlabel('label',properties...)
```

23.51.2 Example

Here is an example of a simple plot with a label on the `x`-axis.

```
--> x = linspace(-1,1);
--> y = cos(2*pi*x);
--> plot(x,y,'r-');
--> xlabel('time');
```

which results in the following plot.



23.52 XLIM Adjust X Axis limits of plot

23.52.1 Usage

There are several ways to use `xlim` to adjust the X axis limits of a plot. The various syntaxes are

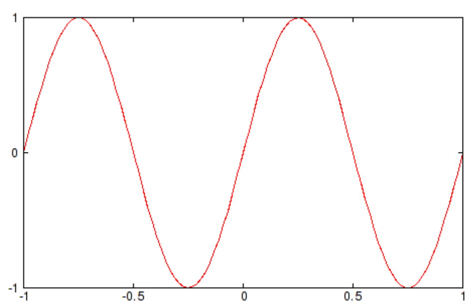
```
xlim
xlim([lo,hi])
xlim('auto')
xlim('manual')
xlim('mode')
xlim(handle,...)
```

The first form (without arguments), returns a 2-vector containing the current limits. The second form sets the limits on the plot to `[lo,hi]`. The third and fourth form set the mode for the limit to `auto` and `manual` respectively. In `auto` mode, FreeMat chooses the range for the axis automatically. The `xlim('mode')` form returns the current mode for the axis (either `'auto'` or `'manual'`). Finally, you can specify the handle of an axis to manipulate instead of using the current one.

23.52.2 Example

```
--> x = linspace(-1,1);
--> y = sin(2*pi*x);
--> plot(x,y,'r-');
--> xlim  % what are the current limits?
ans =
  <double>  - size: [1 2]

Columns 1 to 2
 -1   1
```

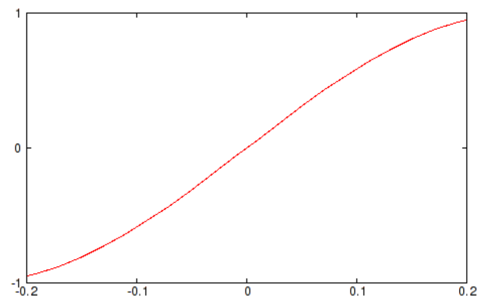
which results in



Next, we zoom in on the plot using the `xlim` function

```
--> plot(x,y,'r-')
--> xlim([-0.2,0.2])
```

which results in



23.53 YLABEL Plot Y-axis Label Function

23.53.1 Usage

This command adds a label to the y-axis of the plot. The general syntax for its use is

```
ylabel('label')
```

or in the alternate form

```
ylabel 'label'
```

or simply

```
ylabel label
```

You can also specify properties for that label using the syntax

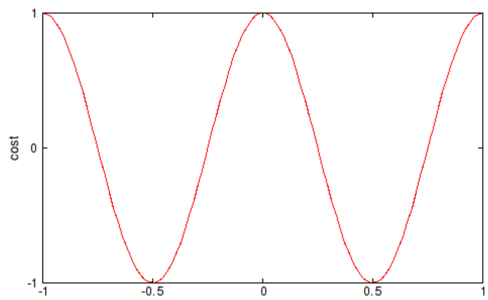
```
ylabel('label',properties...)
```

23.53.2 Example

Here is an example of a simple plot with a label on the y-axis.

```
--> x = linspace(-1,1);
--> y = cos(2*pi*x);
--> plot(x,y,'r-');
--> ylabel('cost');
```

which results in the following plot.



23.54 YLIM Adjust Y Axis limits of plot

23.54.1 Usage

There are several ways to use `ylim` to adjust the Y axis limits of a plot. The various syntaxes are

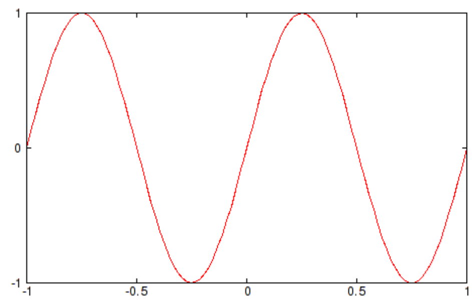
```
ylim
ylim([lo,hi])
ylim('auto')
ylim('manual')
ylim('mode')
ylim(handle,...)
```

The first form (without arguments), returns a 2-vector containing the current limits. The second form sets the limits on the plot to `[lo,hi]`. The third and fourth form set the mode for the limit to `auto` and `manual` respectively. In `auto` mode, FreeMat chooses the range for the axis automatically. The `ylim('mode')` form returns the current mode for the axis (either `'auto'` or `'manual'`). Finally, you can specify the handle of an axis to manipulate instead of using the current one.

23.54.2 Example

```
--> x = linspace(-1,1);
--> y = sin(2*pi*x);
--> plot(x,y,'r-');
--> ylim  % what are the current limits?
ans =
  <double>  - size: [1 2]

Columns 1 to 2
 -1   1
```

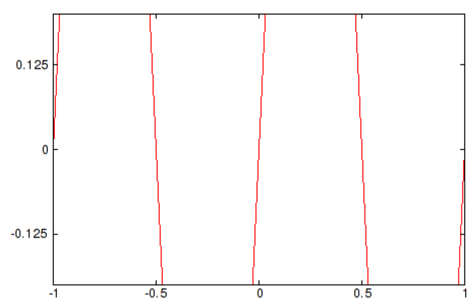
which results in



Next, we zoom in on the plot using the `ylim` function

```
--> plot(x,y,'r-')
--> ylim([-0.2,0.2])
```

which results in



23.55 ZLABEL Plot Z-axis Label Function

23.55.1 Usage

This command adds a label to the z-axis of the plot. The general syntax for its use is

```
zlabel('label')
```

or in the alternate form

```
zlabel 'label'
```

or simply

```
zlabel label
```

Here `label` is a string variable. You can also specify properties for that label using the syntax

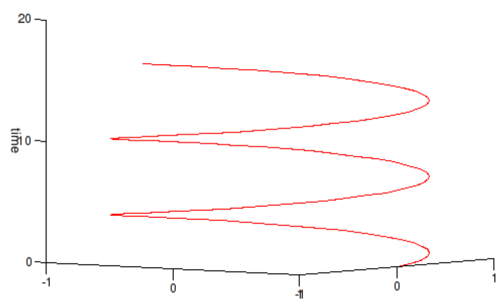
```
zlabel('label',properties...)
```

23.55.2 Example

Here is an example of a simple plot with a label on the `z`-axis.

```
--> t = linspace(0,5*pi);
--> x = cos(t);
--> y = sin(t);
--> z = t;
--> plot3(x,y,z,'r-');
--> view(3);
--> zlabel('time');
```

which results in the following plot.



23.56 ZLIM Adjust Z Axis limits of plot

23.56.1 Usage

There are several ways to use `zlim` to adjust the Z axis limits of a plot. The various syntaxes are

```
zlim
zlim([lo,hi])
zlim('auto')
zlim('manual')
zlim('mode')
zlim(handle,...)
```

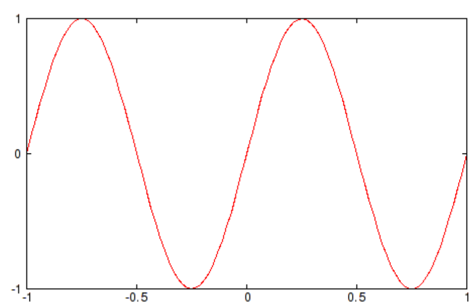
The first form (without arguments), returns a 2-vector containing the current limits. The second form sets the limits on the plot to `[lo,hi]`. The third and fourth form set the mode for the limit to `auto` and `manual` respectively. In `auto` mode, FreeMat chooses the range for the axis automatically.

The `zlim('mode')` form returns the current mode for the axis (either `'auto'` or `'manual'`). Finally, you can specify the handle of an axis to manipulate instead of using the current one.

23.56.2 Example

```
--> x = linspace(-1,1);
--> y = sin(2*pi*x);
--> plot(x,y,'r-');
--> zlim  % what are the current limits?
ans =
  <double>  - size: [1 2]

Columns 1 to 2
 -0.5   0.5
```

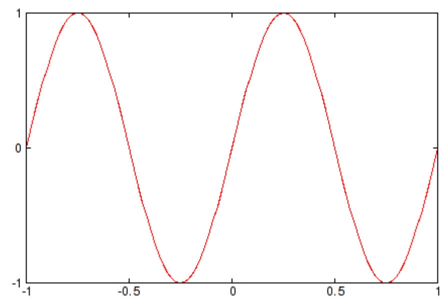
which results in



Next, we zoom in on the plot using the `zlim` function

```
--> plot(x,y,'r-')
--> zlim([-0.2,0.2])
```

which results in



23.57 ZOOM Image Zoom Function

23.57.1 Usage

This function changes the zoom factor associated with the currently active image. It is a legacy support function only, and thus is not quite equivalent to the `zoom` function from previous versions of FreeMat. However, it should achieve roughly the same effect. The generic syntax for its use is

```
zoom(x)
```

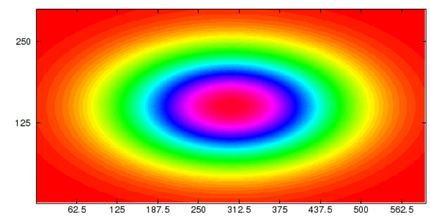
where `x` is the zoom factor to be used. The exact behavior of the zoom factor is as follows:

- `x>0` The image is zoomed by a factor `x` in both directions.
- `x=0` The image on display is zoomed to fit the size of the image window, but the aspect ratio of the image is not changed. (see the Examples section formore details). This is the default zoom level for images displayed with the`image` command.
- `x<0` The image on display is zoomed to fit the size of the image window, with the zoom factor in the row and column directions chosen to fill the entire window. The aspect ratio of the image is not preserved. The exact value of `x` is irrelevant.

23.57.2 Example

To demonstrate the use of the `zoom` function, we create a rectangular image of a Gaussian pulse. We start with a display of the image using the `image` command, and a zoom of 1.

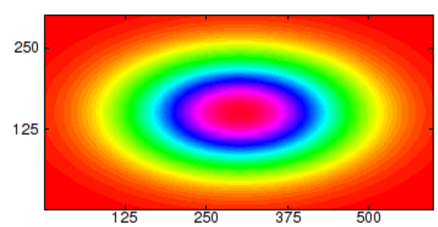
```
--> x = linspace(-1,1,300)'*ones(1,600);
--> y = ones(300,1)*linspace(-1,1,600);
--> Z = exp(-(x.^2+y.^2)/0.3);
--> image(Z);
--> zoom(1.0);
```



At this point, resizing the window accomplishes nothing, as with a zoom factor greater than zero, the size of the image is fixed.

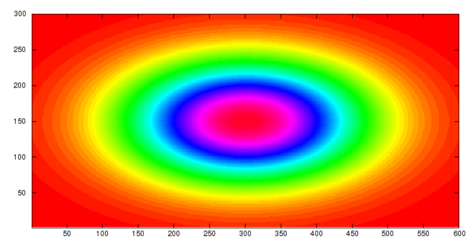
If we change the zoom to another factor larger than 1, we enlarge the image by the specified factor (or shrink it, for zoom factors `0 < x < 1`. Here is the same image zoomed out to 60

```
--> image(Z);
--> zoom(0.6);
```



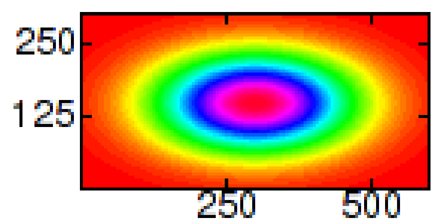
Similarly, we can enlarge it to 130

```
--> image(Z)
ans =
  <uint32>  - size: [1 1]
 100032
--> zoom(1.3);
```



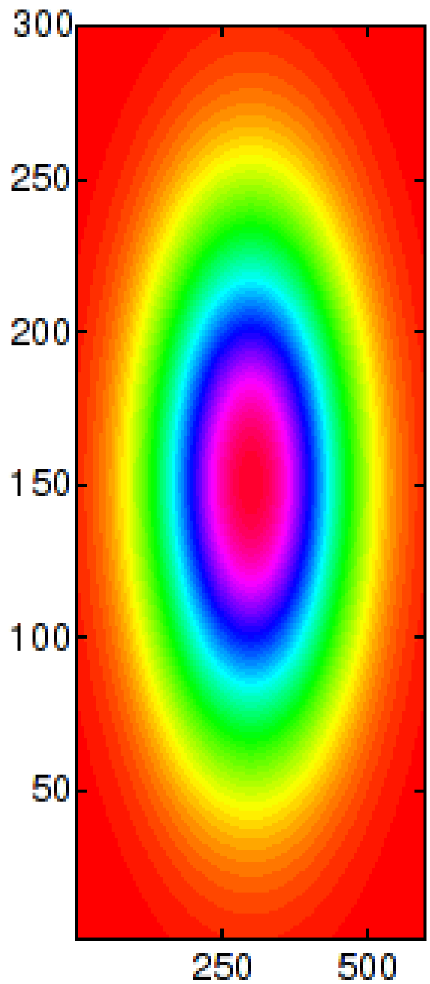
The “free” zoom of `x = 0` results in the image being zoomed to fit the window without changing the aspect ratio. The image is zoomed as much as possible in one direction.

```
--> image(Z);
--> zoom(0);
--> sizefig(200,400);
```



The case of a negative zoom `x < 0` results in the image being scaled arbitrarily. This allows the image aspect ratio to be changed, as in the following example.

```
--> image(Z);
--> zoom(-1);
--> sizefig(200,400);
```


300
250
200
150
100
50
250
500

Chapter 24

Object Oriented Programming

24.1 AND Overloaded Logical And Operator

24.1.1 Usage

This is a method that is invoked to combine two variables using a logical and operator, and is invoked when you call

```
c = and(a,b)
```

or for

```
c = a & b
```

24.2 CLASS Class Support Function

24.2.1 Usage

There are several uses for the `class` function. The first version takes a single argument, and returns the class of that variable. The syntax for this form is

```
classname = class(variable)
```

and it returns a string containing the name of the class for `variable`. The second form of the class function is used to construct an object of a specific type based on a structure which contains data elements for the class. The syntax for this version is

```
classvar = class(template, classname, parent1, parent2,...)
```

This should be called inside the constructor for the class. The resulting class will be of the type `classname`, and will be derived from `parent1`, `parent2`, etc. The `template` argument should be a structure array that contains the members of the class. See the `constructors` help for some details on how to use the `class` function. Note that if the `template` argument is an empty structure matrix, then the resulting variable has no fields beyond those inherited from the parent classes.

24.3 COLON Overloaded Colon Operator

24.3.1 Usage

This is a method that is invoked in one of two forms, either the two argument version

```
  c = colon(a,b)
```

which is also called using the notation

```
  c = a:b
```

and the three argument version

```
  d = colon(a,b,c)
```

which is also called using the notation

```
  d = a:b:c
```

24.4 CONSTRUCTORS Class Constructors

24.4.1 Usage

When designing a constructor for a FreeMat class, you should design the constructor to take a certain form. The following is the code for the sample `mat` object

```
function p = mat(a)
  if (nargin == 0)
    p.c = [];
    p = class(p,'mat');
  elseif isa(a,'mat')
    p = a;
  else
    p.c = a;
    p = class(p,'mat');
  end
```

Generally speaking when it is provided with zero arguments, the constructor returns a default version of the class using a template structure with the right fields populated with default values. If the constructor is given a single argument that matches the class we are trying to construct, the constructor passes through the argument. This form of the constructor is used for type conversion. In particular,

```
   p = mat(a)
```

guarantees that `p` is an array of class `mat`. The last form of the constructor builds a class object given the input. The meaning of this form depends on what makes sense for your class. For example, for a polynomial class, you may want to pass in the coefficients of the polynomial.

24.5 CTRANSPOSE Overloaded Conjugate Transpose Operator

24.5.1 Usage

This is a method that is invoked when a variable has the conjugate transpose operator method applied, and is invoked when you call

```
c = ctranspose(a)
```

or

```
/  c = a'
```

24.6 EQ Overloaded Equals Comparison Operator

24.6.1 Usage

This is a method that is invoked to combine two variables using an equals comparison operator, and is invoked when you call

```
c = eq(a,b)
```

or for

```
c = a == b
```

24.7 GE Overloaded Greater-Than-Equals Comparison Operator

24.7.1 Usage

This is a method that is invoked to combine two variables using a greater than or equals comparison operator, and is invoked when you call

```
c = ge(a,b)
```

or for

```
c = a >= b
```

24.8 GT Overloaded Greater Than Comparison Operator

24.8.1 Usage

This is a method that is invoked to combine two variables using a greater than comparison operator, and is invoked when you call

```
c = gt(a,b)
```

or for

```
c = a > b
```

24.9 HORZCAT Overloaded Horizontal Concatenation

24.9.1 Usage

This is a method for a class that is invoked to concatenate two or more variables of the same class type together. Besides being called when you invoke

```
c = horzcat(a,b,c)
```

when `a` is a class, it is also called for

```
c = [a,b,c]
```

when one of these variables is a class. The exact meaning of horizontal concatenation depends on the class you have designed.

24.10 LDIVIDE Overloaded Left Divide Operator

24.10.1 Usage

This is a method that is invoked when two variables are divided and is invoked when you call

```
c = ldivide(a,b)
```

or for

```
c = a .\ b
```

24.11 LE Overloaded Less-Than-Equals Comparison Operator

24.11.1 Usage

This is a method that is invoked to compare two variables using a less than or equals comparison operator, and is invoked when you call

```
c = le(a,b)
```

or for

```
c = a <= b
```

24.12 LT Overloaded Less Than Comparison Operator

24.12.1 Usage

This is a method that is invoked to compare two variables using a less than comparison operator, and is invoked when you call

```
c = lt(a,b)
```

or for

```
c = a < b
```

24.13 MINUS Overloaded Addition Operator

24.13.1 Usage

This is a method that is invoked when two variables are subtracted and is invoked when you call

```
c = minus(a,b)
```

or for

```
c = a - b
```

24.14 MLDIVIDE Overloaded Matrix Left Divide Operator

24.14.1 Usage

This is a method that is invoked when two variables are divided using the matrix (left) divide operator, and is invoked when you call

```
c = mldivide(a,b)
```

or for

```
c = a \ b
```

24.15 MPOWER Overloaded Matrix Power Operator

24.15.1 Usage

This is a method that is invoked when one variable is raised to another variable using the matrix power operator, and is invoked when you call

```
c = mpower(a,b)
```

or

```
c = a^b
```

24.16 MRDIVIDE Overloaded Matrix Right Divide Operator

24.16.1 Usage

This is a method that is invoked when two variables are divided using the matrix divide operator, and is invoked when you call

```
c = mrdivide(a,b)
```

or for

```
c = a / b
```

24.17 MTIMES Overloaded Matrix Multiplication Operator

24.17.1 Usage

This is a method that is invoked when two variables are multiplied using the matrix operator and is invoked when you call

```
c = mtimes(a,b)
```

or for

```
c = a * b
```

24.18 NE Overloaded Not-Equals Comparison Operator

24.18.1 Usage

This is a method that is invoked to combine two variables using a not-equals comparison operator, and is invoked when you call

```
c = ne(a,b)
```

or for

```
c = a != b
```

24.19 NOT Overloaded Logical Not Operator

24.19.1 Usage

This is a method that is invoked when a variable is logically inverted, and is invoked when you call

```
c = not(a)
```

or for

```
c = ~a
```

24.20 OR Overloaded Logical Or Operator

24.20.1 Usage

This is a method that is invoked to combine two variables using a logical or operator, and is invoked when you call

```
c = or(a,b)
```

or for

```
c = a | b
```

24.21 PLUS Overloaded Addition Operator

24.21.1 Usage

This is a method that is invoked when two variables are added and is invoked when you call

```
c = plus(a,b)
```

or for

```
c = a + b
```

24.22 POWER Overloaded Power Operator

24.22.1 Usage

This is a method that is invoked when one variable is raised to another variable using the dot-power operator, and is invoked when you call

```
c = power(a,b)
```

or

```
c = a.^b
```

24.23 RDIVIDE Overloaded Right Divide Operator

24.23.1 Usage

This is a method that is invoked when two variables are divided and is invoked when you call

```
c = rdivide(a,b)
```

or for

```
c = a ./ b
```

24.24 SUBSASGN Overloaded Class Assignment

24.24.1 Usage

This method is called for expressions of the form

```
a(b) = c, a{b} = c, a.b = c
```

and overloading the `subsasgn` method can allow you to define the meaning of these expressions for objects of class `a`. These expressions are mapped to a call of the form

```
a = subsasgn(a,s,b)
```

where `s` is a structure array with two fields. The first field is

- `type` is a string containing either `'()'` or `'{}'` or `'.'` depending on the form of the call.
- `subs` is a cell array or string containing the the subscript information.

When multiple indexing experssions are combined together such as `a(5).foo{:} = b`, the `s` array contains the following entries

```
s(1).type = '()'  s(1).subs = {5}
s(2).type = '.'   s(2).subs = 'foo'
s(3).type = '{}'  s(3).subs = ':'
```

24.25 SUBSINDEX Overloaded Class Indexing

24.25.1 Usage

This method is called for classes in the expressions of the form

```
c = subsindex(a)
```

where `a` is an object, and `c` is an index vector. It is also called for

```
c = b(a)
```

in which case `subsindex(a)` must return a vector containing integers between `0` and `N-1` where `N` is the number of elements in the vector `b`.

24.26 SUBSREF Overloaded Class Indexing

24.26.1 Usage

This method is called for expressions of the form

```
c = a(b), c = a{b}, c = a.b
```

and overloading the `subsref` method allows you to define the meaning of these expressions for objects of class `a`. These expressions are mapped to a call of the form

```
b = subsref(a,s)
```

where `s` is a structure array with two fields. The first field is

- `type` is a string containing either `'()'` or `'{}'` or `'.'` depending on the form of the call.
- `subs` is a cell array or string containing the the subscript information.

When multiple indexing experssions are combined together such as `b = a(5).foo{:}`, the `s` array contains the following entries

```
s(1).type = '()'  s(1).subs = {5}
s(2).type = '.'   s(2).subs = 'foo'
s(3).type = '{}'  s(3).subs = ':'
```

24.27 TIMES Overloaded Multiplication Operator

24.27.1 Usage

This is a method that is invoked when two variables are multiplied and is invoked when you call

```
c = times(a,b)
```

or for

```
c = a .* b
```

24.28 TRANSPOSE Overloaded Transpose Operator

24.28.1 Usage

This is a method that is invoked when a variable has the transpose operator method applied, and is invoked when you call

```
c = transpose(a)
```

or

```
/  c = a.'
```

24.29 UMINUS Overloaded Unary Minus Operator

24.29.1 Usage

This is a method that is invoked when a variable is negated, and is invoked when you call

```
c = uminus(a)
```

or for

```
c = -a
```

24.30 VERTCAT Overloaded Vertical Concatenation

24.30.1 Usage

This is a method for a class that is invoked to concatenate two or more variables of the same class type together. Besides being called when you invoke

```
c = vertcat(a,b,c)
```

when `a` is a class, it is also called for

```
c = [a;b;c]
```

when one of the variables is a class. The exact meaning of vertical concatenation depends on the class you have designed.

Chapter 25

Bitwise Operations

25.1 BITAND Bitwise Boolean And Operation

25.1.1 Usage

Performs a bitwise binary and operation on the two arguments and returns the result. The syntax for its use is

```
y = bitand(a,b)
```

where `a` and `b` are unsigned integer arrays. The `and` operation is performed using 32 bit unsigned intermediates. Note that if `a` or `b` is a scalar, then each element of the other array is anded with that scalar. Otherwise the two arrays must match in size.

25.1.2 Example

Here we AND some arrays together

```
--> bitand([3 4 2 3 10 12],5)
ans =
  <uint32>  - size: [1 6]

Columns 1 to 6
 1  4  0  1  0  4
```

This is a nice trick to look for odd numbers

```
--> bitand([3 4 2 3 10 12],1)
ans =
  <uint32>  - size: [1 6]

Columns 1 to 6
 1  0  0  1  0  0
```

25.2 BITOR Bitwise Boolean Or Operation

25.2.1 Usage

Performs a bitwise binary or operation on the two arguments and returns the result. The syntax for its use is

```
y = bitor(a,b)
```

where `a` and `b` are unsigned integer arrays. The `or` operation is performed using 32 bit unsigned intermediates. Note that if `a` or `b` is a scalar, then each element of the other array is ored with that scalar. Otherwise the two arrays must match in size.

25.2.2 Example

Here we OR some arrays together

```
--> bitor([3 4 2 3 10 12],5)
ans =
  <uint32>  - size: [1 6]

Columns 1 to 6
  7   5   7   7  15  13
```

This is a nice trick to look for odd numbers

```
--> bitor([3 4 2 3 10 12],1)
ans =
  <uint32>  - size: [1 6]

Columns 1 to 6
  3   5   3   3  11  13
```

25.3 BITXOR Bitwise Boolean Exclusive-Or (XOR) Operation

25.3.1 Usage

Performs a bitwise binary xor operation on the two arguments and returns the result. The syntax for its use is

```
y = bitxor(a,b)
```

where `a` and `b` are unsigned integer arrays. The `xor` operation is performed using 32 bit unsigned intermediates. Note that if `a` or `b` is a scalar, then each element of the other array is xored with that scalar. Otherwise the two arrays must match in size.

25.3.2 Example

Here we XOR some arrays together

```
--> bitxor([3 4 2 3 10 12],5)
ans =
  <uint32>  - size: [1 6]

Columns 1 to 6
  6   1   7   6  15   9
```

This is a nice trick to look for odd numbers

```
--> bitxor([3 4 2 3 10 12],1)
ans =
  <uint32>  - size: [1 6]

Columns 1 to 6
  2   5   3   2  11  13
```