RLAB Reference Manual

Brad Hards
bradh@ee.adfa.oz.au

Copyright Copyright ©1994 The RLAB Reference Manual may be reproduced and distributed in whole or in part, subject to the following conditions:

The RLAB Reference Manual is copyrighted by the author. IT IS NOT IN THE PUBLIC DOMAIN.

- The copyright notice above and this permission notice must be preserved complete on all complete or partial copies.
- Any translation or derivative work of the RLAB Reference Manual must be approved by the author in writing before distribution.
- If you distribute the RLAB Reference Manual in part, instructions for obtaining the complete version of this manual by electronic or physical means must be included, however a means for obtaining a complete version need *not* be provided. Someone elses lack of network connectivity is not your problem.
- Small portions may be reproduced as illustrations for reviews or quotes in other works without this permission notice if proper citation is given.
- The GNU General Public License referenced below may be reproduced under the conditions given within it.

Exceptions to these rules may be granted for academic and other special purposes: write to the author and ask. These restrictions are here to protect the author, not to restrict you as educators and learners.

You can get the latest version of this manual as:

`file://evans.ee.adfa.oz.au/pub/RLaB/Target-linux/ref-manual.tar.gz`

Physical copies can be arranged — contact the Author to arrange this.

All source code in the RLAB Reference Manual is placed under the GNU General Public License, available as: `file://prep.ai.mit.edu/pub/gnu/COPYING`

Publishing this Manual If you are a publishing company interested in distributing this manual, read on.

By the license given in the previous section, anyone is allowed to publish and distribute verbatim copies of the RLAB Reference Manual . You don't need explicit permission for this. However, if you would like to distribute a translation or derivative work based on this manual, you must obtain permission from the author, in writing, before doing so.

You may, of course, sell the RLAB Reference Manual for profit. You are encouraged to do so. Keep in mind, however, that because the RLAB Reference Manual is freely distributable, anyone may photocopy or distribute printed copies free of charge, if they wish to do so.

The author would like to be informed of any plans to publish or distribute the RLAB Reference Manual , to know how this manual is becoming available. It is nice to keep tabs on who is doing what.

Contents

Chapter 1

Introduction

RLAB is an interactive or batch mode matrix-oriented programming language. It is intended for prototyping and other tasks that are not real time. RLAB also serves as a convenient interface to the LAPACK, FFTPACK, and RANLIB numerical libraries from netlib.

Invoking RLAB

A properly installed RLAB can be started by using the following:

`$ rlab RETURN`

where text highlighted like `this` is what you enter and `RETURN` means you have entered the `rlab` command by hitting the `RETURN` key.

RLAB will start with a message similar to

```
Welcome to RLaB. New users type 'help INTRO'
RLaB version 0.99b Copyright (C) 1992, 93, 94 Ian Searle
RLaB comes with ABSOLUTELY NO WARRANTY; for details type 'help WARRANTY'
This is free software, and you are welcome to redistribute it under
certain conditions; type 'help CONDITIONS' for details
>
```

If that didn't work, then RLAB is not set up properly on your machine, and you should talk to your local System Administration ogre, and get them to set it up. If you are the System Administrator ogre, then look at the `README.install` file in the RLAB package. Setting up RLAB is not difficult, however it is beyond the scope of this manual.

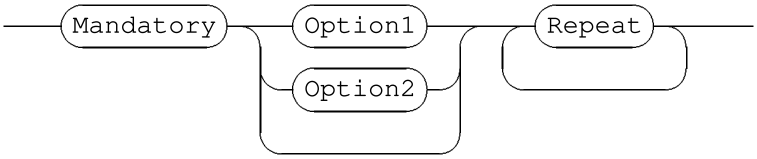
If that *did* work, now is probably a good time to work through the RLAB Primer, which should have come with this manual. The Primer introduces RLAB and is intended to be complementary to this manual. You can get a good feel for the capabilities from the Primer, and the hard details from the Reference Manual. So go to it! The Primer awaits. It doesn't really matter if you don't understand it all, since you can get more information from this manual, but this manual will be a lot easier if you have done the Primer. So give it a try!

Command Line Arguments

When we started RLAB in the last section, we just used the name. That is the way you will normally start RLAB . However, you might also want to change the behavior of RLAB in some cases, which will require you to use an argument to RLAB .

Now is a good time to diverge for a moment. Throughout this manual, I will use 'railroad' diagrams (also known as Bachus-Naur diagrams). You may have seen them used in the past in Pascal books. The basic theme

is that you start in the top-left corner, and work around it. Think of the logic flow as a train. You can't do hairpin turns in a train, and you can't in a railroad diagram either. Here is a simple example:
Demo



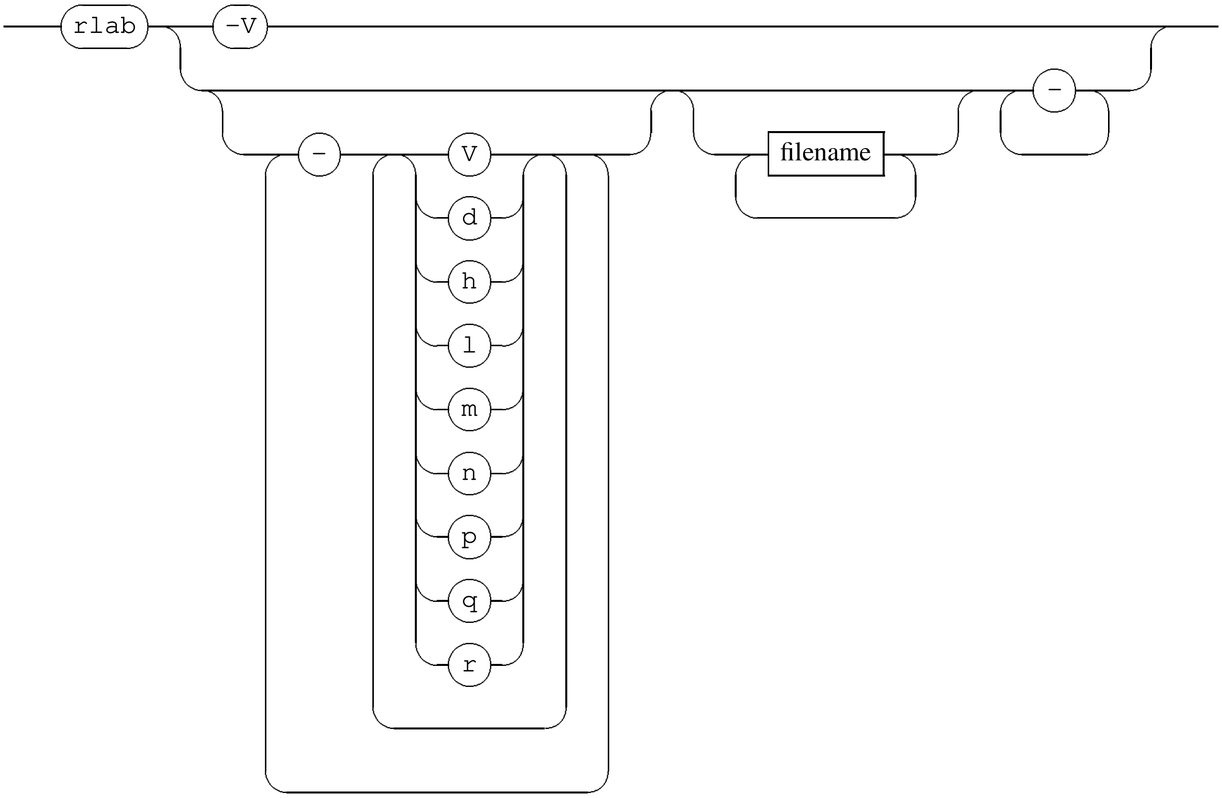
This is what most of the railroad diagrams look like in this manual. The first bit (`Mandatory`) has to occur. The next stage gives you three options. You can do one of three things - `Option1`, `Option2`, or skip that stage all together. The next stage allows you to do that step one or more times. After doing it at least once, you can exit. That was pretty simple wasn't it? If the use of the railroad diagram for something eludes you, have a look for examples of actually using that thing. The railroad diagram is definitive, but examples are usually more obvious.

Now that you know about railroad diagrams, let's use one to show what arguments RLAB can take. If you understand normal Unix notation, then

`rlab [-Vdhlmnpqr] [file(s)] [-]`

is probably a perfectly clear explanation.

Here is the same thing in railroad form:
RLaB



All the things that start with the minus, -, are referred to as the options. An option affects the way RLAB works. The options are mostly for debugging, and they shouldn't normally be required. The options are:

- `-V` which causes a version number to be displayed, and all later arguements to be disregarded. RLABexits after displaying the version number.

`-d` which causes a readable form of the internal stack machine's compiled program to be output to the standard error device. This option should be used in conjunction with `-qln` options. This option is not intended for general use.

`-h` displays the usage message, then exits.

`-l` prevents loading of the RLAB library of rfiles.

`-m` prevents the startup message being displayed.

`-n` prevents line number and file name information from being used in the internal stack machine codes. This option should only be used with the `-dlq` options. This option is not intended for general use.

`-p` prevents rlab from using the specified pager for all output.

`-q` prevents loading of the startup file.

`-r` prevents usage of the GNU readline library for command line editing.

Most of that probably didn't mean a great deal unless you have studied the inner workings of RLAB , but don't worry. You will probably never need to use those options — they are included here only for completeness. If you want more details see Annex A.

The next thing that can occur is one or more filenames. The filenames should contain RLAB code (which we will learn to write later). If you specify several files, it reads from each in turn, as if they were all one file. Note that these files are read after the `.rlab` file and the contents of the RLAB library directory are read. More on those later too.

The final thing that can occur is a dash, `-`, which is only really useful if you specified some files to be read. Normally, RLAB exists after the files have been run. With this option, you get to use RLAB in an interactive mode after the files have been run. Running interactive is the default if you didn't specify any files.

Environmental Variables

When you start RLAB , a whole host of things happen. The first thing that happens is the RLAB program is loaded, and is run. It looks at the options you have specified, and determines what to do next. If you didn't specify the `-q` option, then the the RLAB startup file is run. The default startup file is `/usr/local/lib/rlab/.rlab`, but you can over-ride this by setting an *environmental variable* called `RLAB\_RC0`. Consult the manual for your shell to get more details on how to set up an environmental variable.

The next thing that happens, providing you didn't turn it off by using the `-l` option, is loading the RLAB library. This library is all the files that end in `.r` in the library directory. This is `/usr/local/lib/rlab/rlib` by default, but you can over-ride this one too, by using the environmental variable `RLAB_LIB_DIR`.

After this, RLAB tries to figure out where it should look for help files. The default is `/usr/local/lib/rlab/help`, but the environmental variable `RLAB_HELP_DIR` can be used to change that.

The next thing we do is to find the directory that contains other useful bits of RLAB code, but just aren't used enough to be part of the library (which you will recall is loaded every time RLAB is started). This defaults to `/usr/local/lib/rlab/toolbox`, but like all good defaults, can be over-ridden by the contents of the environmental variable `RLAB_SEARCH_PATH`.

The final environmental variable is the pager. The pager just a program that is used to display output one screen full at a time. The default for this (`more`) is usually fine, but you can use `RLAB_PAGER` or `PAGER` to override it if required. `RLAB_PAGER` has priority.

The most important command

If you have looked through the Primer, then you will be quite familiar with the `help` command. This is the most important command in RLAB , since it can tell you about all the other commands. If you didn't do the Primer, then shame on you, however it is important enough to be said again here.

`help` has two forms. The first gives a list of all the topics you can get help on. All you need to do is type `help`. The exact result depends on your system, but is should be similar to this

```
         BREAK           BUGS        COMMAND   COMMAND_EDIT        COMMENT
       COMPLEX     CONDITIONS   CONTINUATION       CONTINUE       DIVISION
        ERRORS       EXAMPLES          FILES            FOR       FUNCTION
            IF          INTRO       KEYWORDS           LIST    MATLAB_DIFF
      MATRICES      OPERATORS     RELATIONAL           RLAB        SCALARS
        STRING      TRANSPOSE      VARIABLES         VECTOR       WARRANTY
         WHILE            abs           acos            all            any
          asin           atan          atan2        backsub        balance
            cd           ceil           chol          class          clear
         close           conj            cos           cosh            det
          diag          diary            eig          error          exist
           exp         factor            fft         filter           find
         floor         format        fprintf        fvscope         getenv
       getline           help           hess           ifft           imag
           inf            int            inv         issymm         length
          load            log          log10         matrix            max
          maxi        members            min           mini            mod
           nan           norm            ode           ones        plprint
        printf           prod             qr           quit           rand
         rcond           read          readb          readm           real
        replot        reshape          rfile          round         scalar
        schord          schur           show            sin           sinh
          size         sizeof          solve           sort        sprintf
          sqrt          srand        strsplt         strtod            sum
           svd           sylv         system            tan           tanh
           tic            toc           trig           type           what
           who          write         writeb         writem          zeros


/usr/local/lib/rlab/rlib :
         acosh          asinh          atanh       clearall         compan
    complement           cosh          cross        cumprod         cumsum
          diff           disp            dot        epsilon           eval
           eye         finite            fix           hilb          input
       int2str   intersection        isempty          isinf          isnan
      linspace       logspace             lu           lyap           mean
       num2str          pause           plot       printmat           rank
         redit           save            set           sign           sinh
           std           symm           tanh       tmp_file          trace
          tril           triu          union           whos

. :

/usr/local/lib/rlab/toolbox :
      angle     bandred      banner      center        chop         czt
    detrend         erf        expm       faxis     fftplot      fliplr
     flipud        fmin        funm       gamma      hankel       house
     jordan    lagrange        logm        max2      mdsmax        min2
       mret      nmsmax        ode4       ode78      pascal        pinv
  qq_normal         rem         rk4    toeplitz       trapz      window
```

Each of those words is a topic you can get help on. The ones in upper case deal with a concept, and those in lower case deal with a function or command. To get help on a topic, we just type `help` *topic*. Here is an example of how to get help, say on the `zeros` function:

```
> help zeros
zeros:

Syntax: zeros ( nrow, ncol )
        zeros ( A )

Description:

        Zeros returns a matrix with all zero elements. If the
        arguments are two scalars, then zeros returns a matrix with
        dimensions S1xS2.

        If the argument is a MATRIX, then zeros returns a matrix with
        dimensions m[1] by m[2].

        Examples:

        > zeros( 3 , 3 )

        > A = rand(10,4);
        > B = zeros( size(A) )

See Also: size
```

Well that might not have meant anything, but all will become apparent later. The most important help topics to look at when learning are those in upper case, and you might choose to look through them now. The same information (perhaps in a different form) is in later parts of this manual.

Command Line Editting

When you start using RLAB , you will probably find that you make a lot of little errors in syntax. However you don't have to type a whole line again just because you used a semi-colon instead of a comma somewhere. Instead, you can use the editting features provided by RLAB . The basic concept behind this feature is that you can recall previous lines, and modify them, which will produce normal input, just as if you had typed the whole thing in again. This feature is not restricted to just editting the last line — you should be able to go back to any previous line.

To edit a line, the first thing you have to do is recall the line you want to edit. Most commonly this will just be one or more presses of the up-arrow key. If you go past the line you wanted, just use the down-arrow key to compensate. Then you use the left and right arrow keys to select the point where you want to add or delete characters. You add characters by typing them, and delete characters using the Delete or Backspace keys. Characters are added just to the left of the cursor, and are deleted to the left of the cursor. You can then hit the return key (anywhere on the line - it makes no difference), and the whole line is sent to the RLAB interpreter.

However not all terminals have arrow keys, so there is another way, which just uses control keys. They are listed below. For those that start with the caret (`^`) symbol, you hold down the CTRL (Control) key down while typing the following character. For those that start with `ESC-`, you type the ESC (Escape) key, and then type the following character — you don't hold down the ESC key.

These are the getline key bindings:

`^A`	Move cursor to beginning of line
`^B`	Move cursor left (back) one column.
`ESC-B`	Move cursor back one word.
`^D`	Delete the character under the cursor.
`^E`	Move cursor to end of line.
`^F`	Move cursor right (forward) one column.
`ESC-F`	Move cursor forward one word.
`^K`	Kill from cursor to the end of the line.
`^L`	Redisplay current line.
`^N`	Fetches next line from the history list.
`^P`	Fetches previous line from the history list.
`^R`	Search backwards.
`^S`	Search forwards.
`^T`	Swap character under cursor with character to the left.
`^U`	Kill the entire line.
`^Y`	Yank previously killed text back at current location.
`BACKSPACE`	Delete the character left of the cursor.
`DEL`	Delete the character left of the cursor.
`RETURN`	Return the current line.
`TAB`	Jump to next tab stop.
UP-ARROW	Retrieve previous line from history list.
DOWN-ARROW	Retrieve next line from history list.
LEFT-ARROW	Move cursor left (back) one column.
RIGHT-ARROW	Move cursor right (forward) one column.

In addition, there may be other commands, which are listed below.

To make things a little complex, there are three possible ways that command line editting can be set up using RLAB . The first is that it cannot be used. This is normally a result of using RLAB on a weird Operating System, or using the `-r` option at start-up. If the control characters have no effect (and you have suitable text for them to work on), then this is probably the reason.

The second way is that the GNU Readline package has been used. This is very similar to the commands in Emacs. There may be more extensive documentation available using the GNU Emacs *info* command, under *Readline*. The commands added are:

`^_`	Undo the last command.
`ESC-d`	Kill to end of word.
`ESC-DEL`	Delete to start of word.
`ESC-y`	Rotate the kill-ring.
`ESC->`	Beginning of history.
`ESC-<`	End of history.
`ESC-u`	Uppercase word.
`ESC-l`	Lowercase word.
`ESC-c`	Capitalise word.

The third situation is if the normal GetLine package has been used. The commands are similar to Emacs, but this system has a few less features, and is less resource hungry. It adds the following commands:

`^H`	Delete the character left of the cursor.
`^I`	Insert spaces to the next tab stop.
`^J`	Same as hitting RETURN key.
`^M`	Same as hitting RETURN key.
`^O`	Toggle overwrite/insert mode.

Chapter 2

Scalars

Creating Scalars

Scalar Operations

RLAB has a wide range of operators to act upon scalars. The unary operators defined on scalars are:

– Unary negation is the highest priority operator. It is the same as multiplying the expression on the right of the operator by -1.

++ Increments the operand. Operates on the operand to the left of the operator. If the imaginary part of the operand is zero, only the real part is incremented, otherwise both real and imaginary parts are incremented.

– – Decrements the operand. Operates on the operand to the left of the operator. If the imaginary part of the operand is zero, only the real part is decremented, otherwise both real and imaginary parts are decremented.

Lets look at the various binary operations, for `A binop B`:

+ Adds the operands.

– Subtracts the second operand from the first operand.

* Multiplies the operands together.

/ Divides the first operand by the second operand.

\ Divides the first operand by the second operand. This is the same as the right division operator on scalars — it differs only on matrices.

./ Divides the first operand by the second operand. This is the same as the right division operator on scalars — it differs only on matrices.

.\ Divides the first operand by the second operand. This is the same as the right division operator on scalars — it differs only on matrices.

.* Multiplies the operands together. This is the same as the `*` operator on scalars — it differs only on matrices.

^ `A^B` raises `A` to the `B` power.

.^ `A.^B` raises `A` to the `B` power. This is the same as the normal power operator on scalars — it differs only on matrices.

The operators that are denoted by two symbols should not have any white space (or any other characters) between the two symbols.

Relational Operations

RLAB relational, equality and logical operators return 1 if the expression is true, and 0 if the expression is false.

The RLAB relational operators are

$<$ true if the expression on the left hand side is less than the expression on the right hand side.

$<=$ true if the expression on the left hand side is less than or equal to the expression on the right hand side.

$>$ true if the expression on the left hand side is greater than the expression on the right hand side.

$>=$ true if the expression on the left hand side is greater than or equal to the expression on the right hand side.

The relational operators all have the same precedence. If the expression on either side is complex, then comparison is done on magnitudes.

RLaB equality operators are ==, which is the test for equality, and `!=`, which is the test for non-equality. The equality operators have the same precedence, and are just below the relational operators, in terms of precedence.

RLAB logical operators are `&&`, which is logical and, and `||`, which is logical or. Logical AND operation has higher precedence than logical OR. Both logical operators are lower in precedence than the equality operators.

The logical operators evaluate left-to-right, however evaluation does not stop as soon as the result is known, as in C. Full evaluation in required because the operands are not restricted to being simple scalars quantities. Cases where the operands are matrices requires that full operand evaluation occur.

Scalar Functions

Random numbers

One of the most flexible features in RLAB is the way it can generate random numbers. There are only two functions used in random number creation — `rand`, and `srand`.

`srand` (See Page 209), is used to set the seed value that the numbers are produced from. Using the same seed value each time will produce the same sequence of random numbers. You should use `srand(``clock'')` if you want a unique sequence.

`rand` is the function that actually produces the random values. You can choose many different distributions, though the uniform and normal distributions are likely to be of most use. A complete list of distributions is given on Page 178.

Chapter 3

Matrices

This chapter is an introduction to the matrix data type. Matrices are the most commonly used data type in RLAB and there are many powerful operations and functions you can perform on them. A matrix is generally a two dimensional array of scalars, although RLAB also supports string matrices, see Chapter 4. All numeric operations and functions work for any combination of real and complex operands or arguments.

The elements inside a matrix are refered to by the number of the row and column they are in. Rows are specified first and go across, and are numbered from top to bottom. Columns go down, and are numbered from left to right.

Matrix Creation

The `[` and `]` operators are used to create and access a matrix. For example, to create a matrix at the command line, you would just type:

```
> m = [ 1, 2, 3; 4, 5, 6; 7, 8, 9 ]
 m =
 matrix columns 1 thru 3
        1          2          3
        4          5          6
        7          8          9
```

The rows of the matrix are delimited with '`;`' and the elements of each row are delimited with '`,`'. If you want a complex matrix you enter it using the normal notation of real and imaginary parts. Here is an example:

```
> z= [ 1+2i, 2+3j, 3 + 4i;
>      4 -5i, -5+6j, -6 -7j ]
 z =
 matrix columns 1 thru 3
                  1 + 2i                  2 + 3i                  3 + 4i
                  4 - 5i                 -5 + 6i                 -6 - 7i
```

This example shows a few things that are pretty important. You can use either `i` or `j`, but you can't leave spaces between the imaginary part and the letter. Spaces are generally fine everywhere else.

While the previous method for matrix creation is quite convenient, often you will need specific types of matrices, such as an identity matrix, a matrix of random values, or a Hilbert matrix. There are RLAB functions to produce these and many other types of matrices. All functions are described in Chapter 9, however those that are likely to be of particular interest for creating matrices are:

compan generates the companion matrix of its argument

diag diagonalises its argument

eye produces an identity matrix

hilb produces a Hilbert matrix

linspace and logspace produce spaced vectors

ones produce a matrix with all elements 1.

rand generates random matrix

reshape changes the structure of a matrix

zeros produce a matrix with all elements 0.

Another useful method for creating a matrix is to read the values from a file. RLAB provides several ways of doing this. The two most popular are `readm()` and `read()`.

`read()` reads a matrix that has been written with the RLAB `write()` function (See Section 9 and Section 9). Here is a contrived example - you would normally `write` it out in one session, and `read` it again in a later session. You should also realise that `read` and `write` are not restricted to a storing and retrieving a single variable, or even a single type of variable, in a file. You can store several different types of variables in each file, according to your needs.

```
> z = [ 3 , 4; -4-3j , 786 ]
 z =
                   3 + 0i                   4 + 0i
                  -4 - 3i                 786 + 0i
> who()
eps  pi   z
> write( "save_z.dat" , z )
        1
> // We have now written out the matrix ‘z’ to the file ‘save_z.dat’
> clear(z);
> who()
eps  pi
> // The matrix ‘z’ is now gone from the symbol table
> read( "save_z.dat" )
        1
> who()
eps  pi   z
> // And now it has been read back in.
> diary()
```

`readm()` (See Section 9) reads a text file that contains white-space separated columns of numbers. This is useful if you are trying to use the output of other programs. If you want to, you can also write out a matrix in the same format, using the `writem` function. If you don’t like the way the matrix is organised, the `reshape()` function (See Section 9) is a good way to restructure it.

```
> a = [ 1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12]
 a =
        1          2          3          4
        5          6          7          8
        9         10         11         12
> // Now we write the matrix to a file.
> writem("matrix_text", a)
```

```
        1
> clear(a)
        1
> who()
eps  pi
> // And get it back - store it in a different variable this time
> b = readm("matrix_text")
 b =
        1          2          3          4
        5          6          7          8
        9         10         11         12
> // But we want those elements organised as two rows and six columns
> c = reshape(b, 2,6)
 c =
 matrix columns 1 thru 6
        1          9          6          3         11          8
        5          2         10          7          4         12
```

It appears that things have gone horribly wrong in that last example. However this is expected behavior. The matrix is stored column by column, and if you run down each column in each matrix, the order is the same in each matrix. The transposition operator might come in handy if you want it to work by rows, as shown here:

```
> d = reshape(b', 2,6)
 d =
 matrix columns 1 thru 6
        1          3          5          7          9         11
        2          4          6          8         10         12
> e = (reshape(b',6,2))'
 e =
 matrix columns 1 thru 6
        1          2          3          4          5          6
        7          8          9         10         11         12
```

Sometimes when you have a problem that is represented as two matrices, you would like to combine them to make a single matrix. This is a pretty simple task as soon as you realise that there is nothing that makes the things inside the square brackets have to be scalars — they can also be strings and other matricies. We will look at using matricies here, and deal with strings later.

When trying to tack several matricies together to make a new matrix, or using a combination of scalars and matricies to make a new matrix, you should think of the matricies as being equivalent to how they would be represented if you had to type them yourself. So imagine that each row is terminated with a semicolon and each element seperated by a comma. Then fill in commas and semicolons between the elements to make up the new matrix. Here are some examples:

```
> a = [1, 2,3; 4,5,6]
 a =
        1          2          3
        4          5          6
> b = [ 7, 8, 9; 10, 11, 12]
 b =
        7          8          9
       10         11         12
> c = [a,b]
 c =
 matrix columns 1 thru 6
```

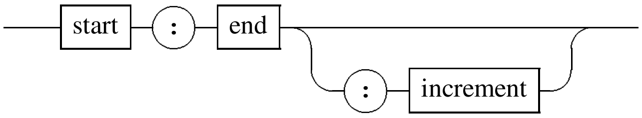
```
        1          2          3          7          8          9
        4          5          6         10         11         12
> d = [a;b]
 d =
        1          2          3
        4          5          6
        7          8          9
       10         11         12
> e = [ -2, -1 ,0; a]
 e =
       -2         -1          0
        1          2          3
        4          5          6
```

Vector Creation

Although there is no distinct vector type in RLaB, you can pretend that there is. If your algorithm or program does not need two dimensional arrays, then you can use matrices as one dimensioned arrays.

When using vectors, or single dimension arrays, row matrices are created. The simplest way to create a vector is with the '`:`' operator(s):

Vector



The first operand specifies the starting value, the second operand specifies the last value. The optional third operand can be used to specify an increment. If this is not specified, the default increment is 1. Here are some examples:

```
> c = 1:4
 c =
        1          2          3          4
> d = 1:3:0.5
 d =
        1        1.5          2        2.5          3
> e = 1:3:0.6
 e =
        1        1.6        2.2        2.8
```

The other way to generate a vector is to simply specify it using the matrix form with one of the dimensions set to 1. This method allows you to generate a vector containing any elements, not just an evenly spaced sequence. Just remember that RLaB doesn't know anything about vectors — they are just a matrix where one of the dimensions is 1. Some functions do something slightly different if they come across a matrix that is the vector form, but they are really just a notational convenience, not a fundamental change, and anything that applies to a matrix also applies to something that we treat as a vector.

Matrix Attributes

The attributes of a matrix, such as how many rows it has, are accessible in several ways. All attributes are accesible through function calls, for example:

```
> my_matrix = [ 1, 0.333, 0.367, 0.24;
>               6, -0.3, 0.9, 3];
> show(my_matrix)
   name:      my_matrix
   class:     num
   type:      real
     nr:      2
     nc:      4
> size(my_matrix)
        2          4
> length(my_matrix)
        4
> class(my_matrix)
num
> type(my_matrix)
real
> name(my_matrix)
my_matrix
```

You can learn more about each of those functions by refering to the Function Reference, or by using the on-line help.

Matrix attributes are also accessible via a shorthand notation:

```
> // Number of rows
> my_matrix.nr
        2
> // Number of columns
> my_matrix.nc
        4
> // Number of elements
> my_matrix.n
       8
> // Class of variable
> my_matrix.class
num
> // Type of contents
> my_matrix.type
real
```

Referencing a Matrix

It doesn't matter how a matrix was created - using the vector or matrix methods, there are two ways to access it. You can either specify an row and column, or you can specify a particular element. For example:

```
> a = rand(3,4)
 a =
 matrix columns 1 thru 4
        1      0.333      0.665      0.167
    0.975     0.0369     0.0847      0.655
    0.647      0.162      0.204      0.129
> a[1,4]
    0.167
> a[4]
```

```
    0.333
> v = 1:4
 v =
 matrix columns 1 thru 4
        1          2          3          4
> v[1;3]
        3
> v[2]
        2
```

As you can see, matrices are stored internally in a column-wise fashion. To force a matrix to it's internal form you can use the '`[:]`' operator:

```
> a = rand(2,3)
 a =
 matrix columns 1 thru 3
        1      0.647     0.0369
    0.975      0.333      0.162
> a [ : ]
 matrix columns 1 thru 1
        1
    0.975
    0.647
    0.333
   0.0369
    0.162
```

In addition to accessing single elements, we can also access partial rows and/or columns of a matrix. We use the '`;`' symbol to delimit row and column indicies, and the '`,`' symbol to delimit individual row or column indicies. To reference an entire row or column, we leave out the column or row index respectively:

```
> g = rand(3,4)
 g =
        1      0.333      0.665      0.167
    0.975     0.0369     0.0847      0.655
    0.647      0.162      0.204      0.129
> g[3;]
    0.647      0.162      0.204      0.129
> g[;2]
    0.333
   0.0369
    0.162
```

To reference a sub-matrix, we just specify which rows and columns are to be extracted:

```
> g[2; 3,4]
   0.0847      0.655
```

As stated previously, any expression that evaluates to a matrix can have it's elements referenced. For example, the `size()` function returns a two element matrix, where the first element contains the number of rows in the argument, and the second element contains the number of columns. For example:

```
> size(g)[2]
        4
```

Assignment

Just as we can read from parts of a matrix, we can also write to part of a matrix. You have seen assignment of a whole matrix to a variable in previous examples — now lets look at partial assignment. We can re-assign single elements, or groups of elements. For single elements, we just specify a reference to that element (either row and column or vector notation — it doesn't matter), and equate it to the new value. For a group of elements, we have to specify a range, and equate that to a matrix of equal size. Here are some examples:

```
> yellow = rand(3,3)
 yellow =
    0.167       0.91      0.265
    0.655      0.112        0.7
    0.129      0.299       0.95
> yellow[2;2] = 967
 yellow =
    0.167       0.91      0.265
    0.655        967        0.7
    0.129      0.299       0.95
> yellow[2,3;1,2] = [100, 200; 300, 400]
 yellow =
    0.167       0.91      0.265
      100        200        0.7
      300        400       0.95
```

Matrix Operations

The usual mathematical operators (eg. `+, -, *, /`) operate on matrices as well as scalars. Lets look at the various binary operations, for `A binop B`:

+ Does element-by-element addition of two matrices. The row and column dimensions of `A` and `B` must be the same, unless either `A` or `B` is a 1-by-1 matrix; in this case a scalar-matrix addition operation is performed.

– Does element-by-element subtraction of two matrices. The row and column dimensions of both `A` and `B` must be the same, unless either `A` or `B` is a 1-by-1 matrix; in this case a scalar-matrix subtraction operation is performed.

***** Performs matrix multiplication on the two operands. The column dimension of `A` must match the row dimension of `B`, unless either `A` or `B` is a 1-by-1 matrix; in this case a scalar-matrix multiplication is performed.

/ Performs matrix "right-division" on it's operands. The matrix right-division (`A/B`) can be thought of as `A*inv (B)`. The column dimensions of `A` and `B` must be the same. Internally right division is the same as "left-division" with the arguments transposed.

$$A/B = (B^T \setminus A^T)^T$$

The exception to this dimension rule occurs when `B` is a 1-by-1 matrix; in this case a matrix-scalar divide occurs.

Additionally, RLaB has several other operators that function on matrix operand(s).

.* Performs element-by-element multiplication on it's operands. The operands must have the same row and column dimensions, unless either `A` or `B` is a 1-by-1 matrix.

./ Performs element-by-element division on it's operands. The operands must have the same row and column dimensions, unless either `A` or `B` is a 1-by-1 matrix.

\ Performs matrix "left-division". Given operands `A\B`, matrix left division is the solution to the set of equations $Ax = B$. If B has several columns, then each column of x is a solution to `A*x[;i] = B[;i]`. The row dimensions of `A` and `B` must agree.

.\ Performs element-by-element left-division. Element-by-element left-division is provided for symmetry, and is equivalent to `B./A`. The row and column dimensions of `A` and `B` must agree, unless either one is a 1-by-1 matrix.

^ `A^B` raises `A` to the `B` power. When `A` is a matrix, and `B` is an integer scalar, the operation is performed by successive multiplications. When `B` is not an integer, then the operation is performed using `A`'s eigenvalues and eigenvectors. The operation is not allowed if `B` is a matrix.

.^ `A.^B` raises `A` to the `B` power in an element-by-element fashion. Either `A` or `B` can be matrix or scalar. If both `A` and `B` are matrices, then the row and column dimensions must agree.

' This unary operator performs the matrix transpose operation. `A'` swaps the rows and columns of A. For a matrix with complex elements a complex conjugate transpose is performed.

.' This unary operator performs the matrix transpose operation. `A.'` swaps the rows and columns of A. The difference between `'` and `.'` is only apparent when `A` is a complex matrix; then `A.'` does not perform a complex conjugate transpose.

There are several details that are very important to note:

- The operators that are denoted by two symbol should not have any white space or any other character between the two symbols.
- The expression `2./A` is **not** interpreted as `2. /A`. RLaB is smart enough to group the period with the `/`.

Matrix Relational Operations

Just as we can perform relational operations on scalars, we can also do matrix relational operations. When we do a relational test on two matrices, such as `A == B`, the operands must be the same dimensions, or one of them must be a 1-by-1 matrix. The result of a matrix relational test is a matrix the same size as the operands filled with ones and zeros according to the result of an element-by-element test. A one in a particular location means the test was true for the pair of elements in those locations in the operands. For example:

```
> a = [1, 2, 3; 4, 5, 6; 7, 8, 9]
 a =
 matrix columns 1 thru 3
        1          2          3
        4          5          6
        7          8          9
> b = a'
 b =
 matrix columns 1 thru 3
        1          4          7
        2          5          8
        3          6          9
> a == b
 matrix columns 1 thru 3
        1          0          0
        0          1          0
        0          0          1
> a >= 5
```

```
 matrix columns 1 thru 3
        0          0          0
        0          1          1
        1          1          1
```

RLAB if-tests do not accept matrices. The `any()` and `all()` functions can be used in combination with relational and logical tests to conditionally execute statements based upon matrix properties.

The function `any()` returns true if any of the elements of it's argument are non-zero. The function `all()` returns true if all of the elements of it's argument are non-zero. The `any()` function is called on the output of another `any()` call because the `any()` function returns a vector when passed a matrix that is not already a vector.

Functions

You have already seen some of the functions in use. There are for basic types of functions that can operate on matricies in RLABThey are:

Scalar Functions: These functions operate on scalars, and treat matrices in an element-by-element fashion. Some examples are:

abs	exp	floor	round
cos	sin	tan	ceil
sqrt	real	imag	conj
isnan	int		

Vector Functions: These functions operate on vectors, either row-vectors (1-by-n) or column vectors (m-by-1), in the same fashion. If the argument is a matrix with both dimensions > 1 then the operation is performed in a column-wise fashion. Some examples are:

sum	prod	mean	max
min	fft	sort	any

When using a vector oriented function, like `max()`, on a general matrix, it is possible to obtain scalar quantities. For example the maximum value in a matrix can be obtained using `max (max (a))`. The first call to `max()` returns a vector of the maximum values from each column, and the second call to `max()` returns the maximum value in the matrix.

Matrix Functions: These functions operate on matrices as a single entity. Some examples are:

balance	chol	det	eig
hess	inv	lu	norm
pinv	qr	rank	rcond
reshape	solve	svd	symm

Chapter 4

Strings

A string is a sequence of text characters enclosed in double quote characters: `"`. Single quotes are not sufficient — forward quoted: ` have no meaning, and backward quotes: `'` are used to indicate matrix transpose.

String Creation

Strings are pretty simple to create — you just assign a sequence of text characters to a variable.

```
> str = "Sample String"
Sample String
```

The `show` command reveals the attributes associated with each string. The information can also be obtained using the normal shorthand method.

```
> show ( str )
   name:      str
   class:     string
    type:
     nr:      1
     nc:      1
> str.class
string
> str.l
          13
```

You can also put escape characters in strings. The following escape codes are supported:

- `\n` which produces a newline
- `\t` which produces a horizontal tab
- `\f` which produces a form feed
- `\b` which produces a backspace
- `\r` which produces a carriage return
- `\a` which produces an alert (bell)
- `\v` which produces a vertical tab
- `\\` which produces a backslash
- `\'` which produces a single quote
- `\"` which produces a double quote

String Matrices

Strings can be used to form matrices, in the same way that numeric values can form a numeric matrix:

```
> StrMat = [ "A string", "Another String";
>            "A third string", "The last string" ];
> StrMat
 StrMat =
A string         Another String
A third string   The last string
> show(StrMat)
   name:      StrMat
   class:     string
   type:
     nr:      2
     nc:      2
```

As shown here, the elements of a string matrix do not need to be the same length. However scalar strings cannot be mixed with numeric scalars in the same matrix — a list is the only variable capable of making up a heterogeneous collection.

String Operations

Relational Tests

Any of the normal relational tests can be applied to strings. Two strings are considered equal if they have the same contents and the same length. When trying to decide if one string is less than another, RLAB tests each character position in the string in turn. As soon as they differ, the word containing the character that occurs first in the character set is considered the lesser. Non-existent characters are less than any other character. Here are some examples that make it clearer:

```
> "abc" == "abcd"
        0
> "dabc" == "abcd"
        0
> "dabc" < "abcd"
        0
> "abc" == "abcd"
        0
> "abc" == "abc"
        1
> "abc" == "abcd"
        0
> "abc" < "abcd"
        1
> "dbc" > "abcd"
        1
```

Other Operations

Strings can be copied using the normal assignment operator, that is =. Two strings may be concatenation (joined) by using the + operator. These are the only operations defined on strings.

```
> string1 = "Any old thang"
```

```
Any old thang
> string2 = string1 + "Some more old things"
Any old thangSome more old things
```

This example shows something you should be aware of — the concatenation operator doesn't add any spaces. If you want them, they have to be added as part of the arguments.

String Functions

`strsplt()` is a useful string function. It splits up the argument string and returns a row matrix that contains a single character string as each element. The resulting matrix has as many columns as the input argument had characters. For example:

```
> str = "Yet another string"
Yet another string
> str2 = strsplt(str)
 str2 =
Y  e  t     a  n  o  t  h  e  r     s  t  r  i  n  g
> show(str2)
   name:      str2
   class:     string
   type:
     nr:      1
     nc:      18
```

Formatted Strings

There are three functions that you can use to produce formatted character strings. They are `printf()`, `fprintf()` and `sprintf()`. See pages 170, 102 and 206 respectively have detailed descriptions of the arguments these functions take, however they are described here briefly.

`printf()` is the least general of the functions. However it is quite easy to use, and provides an introduction to the other two functions. `printf` prints a formatted string to standard output. The return value is the number of characters written. Since RLAB often has scalar variables that you want to incorporate into strings, you can use the normal C language formatting options. These are documented, with examples, on Page 170.

`fprintf()` is an exceptionally versatile command, as it allows you to send formatted strings to files and processes. The syntax for this function is given on Page 102. It also takes C language formatting options, and returns the number of characters. Using `stdout` as the filename is the same as using `printf` — you may also find using `stderr` is useful.

`sprintf()` is used to generate a formatted string variable. This function also takes C language formatting options and returns the number of characters in the string produced. Page 206 has further details, with examples.

Chapter 5

Conditional and Looping Constructs

if statement

The `if` statement is very similar to the C if statement.

If

Note that the statements executed inside both the sections of the body can include other `if` statements, as shown below.

Example

```
> if (1+2 <= 4)
  {
  a = 6
  else
  a = -6
  }
 a =
        6
> if (1+2 < 3)
  {
  a = 6
  else
  a = -6
  }
 a =
       -6
> if( class(a) == "num")
  {
  b = tan(a)
  c = 2+3-4
  else
  a = 0
```

```
  }
 b =
    0.291
 c =
        1
> if ( 0 )
  {
  b = " false "
  else if (c < (d = 2))
  {
  e = -d
  else
  e = d
  }}
 e =
       -2
```

for loop

The RLaB for statement is NOT similar to the C for statement. The correct usage is best shown by example...

Example:

```
> for(i in 1:4) {
>   for(j in 1:5) {
>     m[i;j] = i+j;
>   }
> }
```

The above shows a nested set of for statements. i and j are automatically initialized to have the values of the vector expressions '1:4' and '1:5'. Each loop proceeds until i and j have been assigned each element of their respective vector expressions.

The vector in the for statement can be any expression that evaluates to a SCALAR or a MATRIX. If the expression evaluates to a SCALAR the body of the for statement is executed ONCE. If the expression evaluates to an empty-matrix ('[]'), then the loop is not executed at all.

The for statement can be also be used to loop through the elements of a LIST.

Example:

```
> xlist = << Mass = sqrt(200); Inertia = eye(3,3); xdot = [1,2,3] >>
   Inertia      Mass         xdot
> for( i in members(xlist) )
  {
    xlist.[i]
  }
```

```
 Inertia =
 matrix columns 1 thru 3
           1           0           0
           0           1           0
           0           0           1
 Mass =
       14.14
 xdot =
 matrix columns 1 thru 3
   1           2           3
```

while loop

The RLaB while statement is similar to the C while statement.

Example:

```
> i=1;
> while(i<10)
  {
    a[i] = 1.3*i;
    i++;
  }
> a
 a =
 matrix columns 1 thru 5
         1.3         2.6          3.9          5.2          6.5

 matrix columns 6 thru 9
         7.8         9.1        10.4        11.7
```

```
while(0) { "this will never be executed" }
```

If the conditional evaluates to zero then the loop is not executed, if it evaluates to anything other than zero it will be executed until the condition is zero.

break statement

The break statement functions similar to the C break. The break statement provides an early exit from the enclosing for, or while loop. For example:

```
> while ( i < 100 ) {
>   if(i == 10) { break; }
>   i++;
> }
> i
 i =
          10
```

```
//
// OR
//

> i=0;j=0;
> while (i < 5) { while (j < 5) { if (j == 3) { break } j++; } i++; } i j
 i =
          5
 j =
          3
```

RLAB if tests do not accept matrices. The `any()` and `all()` functions can be used in combination with relational and logical tests to conditionally execute statements based upon matrix properties.

The function `any()` (See page 52 returns 1 if any of the elements of its argument are non-zero. The function `all()` (See page 51 returns 1 if all of the elements of it's argument are non-zero. The `any()` function is often called on the result of another `any()` call because the `any()` function returns a vector when passed a matrix that is not already a vector.

Chapter 6

Lists

This chapter assumes knowledge of the scalar, matrix and string variable types. A list is a convenient way to group together different types of variables, and treat them as one variable. This is very similar to the way a Pascal record, or C structure is made up.

A nice thing about lists is that they only contain objects that are explicitly installed, regardless of the index values. For instance, a list with index values of `1` and `100` will only contains two items, the elements for the `1` and `100` indices, no more. Furthermore, lists can be more efficient than appending rows or columns onto matrices, since the memory management overhead is less.

Lists can also be a convenient way to have an array that indexes from zero. This may not be as efficient as using a matrix, though if the problem is expressed more clearly, then a list may be appropriate.

List Creation

A list can be created either explicitly or implicitly. Generally, you will use the implicit method, but we will look at both methods. Also, note that the two methods are not mutually exclusive. In particular, note that it is quite legitimate to create a list explicitly, then add more elements implicitly.

Explicit Creation

Creating a list explicitly is common is user defined functions (covered in the next chapter), where you calculate the various results, then make a list that containing them.

The explicit list creation operators are << and >>. The most basic operation is to create an empty list. Here is an example of creating a list (called `mylist`), with no elements:

```
>  mylist = << >>
        <<>>
```

More useful is creating a list with some elements already in it. You use the same operators, with the names of the variables you want to put into the list within the the list operators. A semicolon should be used to seperate the variables. Here is an example:

```
> a = "a string"
a string
> b = 23.5
 b =
     23.5
> c = rand(2,2)
 c =
        1       0.647
    0.975       0.333
```

```
> heterogenous = << a ; b ; c >>
   1            2            3
```

Notice how the variables are indexed using scalars. Indexing will be discussed later, but suffice for now to say that the index is how you get the induvidual elements back out of the list. The index is often more useful if you use strings that add some meaning to the values contained within the list. You do this by assigning the values to a string label, as shown here:

```
> a = "a string"
a string
> b = 23.5
 b =
     23.5
> heterogenous2 = << String=a ; Weight=b ; Product=rand(3,3) >>
   Product      String       Weight
```

The labels within a list are only visible in relation to that list. That is, they don't appear in the global symbol table. Also, note that the values that are being copied into the list do not neccessarily have to exist before the list is created. The example above shows that you can create the contents at the same time this list is created.

Implicit Creation

Implicit creation is most commonly used at the command line, as it is easier and takes less forward planning. The basic idea is that you can just add an element to a list without regard to whether that list actually exists. If the list doesn't exist, it will be created, and then the element will be added.

Let's look at an example:

```
> who()
eps  pi
> mylist.vect = 1:5
   vect
> mylist.m = rand(2,3)
   m            vect
> mylist.stringval = "one point four two"
   m            stringval    vect
```

In the explicit method, we saw how it was possible to assign elements without indicies, and have RLAB supply scalar indicies automatically. There isn't any way to do this with the implicit list creation method, however it is possible to get scalar indicies if you want them. To do so, you have to use square braces, `[` and `]`, around the scalar index. Carrying on from the previous example, if you want another element with the scalar index `5`, you would do the following.

```
> mylist.[5] = 4:6:0.5
   5            m            stringval    vect
```

However the braces are good for much more than simple scalar assignment. Those braces are actually forcing evaluation of their contents, and then convert the result to a string. Now evaluation of a scalar constant is pretty simple, but here are more complex examples, again following on from previous examples:

```
> e=5
 e =
        5
> mylist.[e+sqrt(4)] = 4.56
   5            7            m            stringval    vect
> stringmat = [ "string1"; "string2" ; "string3" ]
```

```
 stringmat =
string1
string2
string3
> mylist.[stringmat[2]] = " more characters"
   5            7            m            string2      stringval
   vect
```

Although there is considerable power in using scalar indicies, it is strongly recommended that you use descriptive strings unless you are planning on using that power.

List Indexing

To reference a list member, two methods are available. Both are a lot like the implicit list creation, just without the equals sign and right hand side.

The first method is of the form *listname* **.** *index* . It interprets *index* as a character string, and uses that string as an index to the list specifies by *listname*.

The second method uses the square braces, `[` and `]`, and forces evaluation of the contents of the braces as a string, and then uses the resulting string as an index to the list. The expression inside the braces must evaluate to a scalar or string. This method is required if you used the explicit creation method, and did not assign indicies.

```
> // some equivalent methods of getting the first element
> mylist.[5]
 5 =
        4        4.5          5        5.5          6
> e=5
 e =
        5
> mylist.[e]
 5 =
        4        4.5          5        5.5          6
> mylist.[2+sqrt(9)]
 5 =
        4        4.5          5        5.5          6
> // some more equivalents
> mylist.stringval
one point four two
> mylist.["stringval"]
one point four two
> mylist.["stri" + "ngval"]
one point four two
```

Functions returning lists

We have already alluded to the fact that some functions return lists. This is most common when the return value is made up of different types, or is made up of matrices with different dimensions. An example is the `eig` function, which returns a list containing two matricies, as shown here:

```
> a= rand(4,4)
 a =
    0.665      0.655      0.299     0.0918
   0.0847      0.129      0.265      0.902
    0.204       0.91        0.7       0.96
```

```
    0.167      0.112       0.95       0.915
> a_eig = eig(a)
   val          vec
> show(a_eig.val)
   name:       val
   class:      num
   type:       complex
     nr:       1
     nc:       4
> show(a_eig.vec)
   name:       vec
   class:      num
   type:       complex
     nr:       4
     nc:       4
```

However sometimes you don't want all the information. For instance, we might want to conduct a test on the eigenvalues. So we can throw away the eigenvectors, and just look at the eigenvalues. We can do this using either method, as shown here:

```
> eigenvalues1 = eig(a).val
 eigenvalues1 =
 matrix columns 1 thru 3
              0.554 + 0i               2.15 + 0i         -0.148 + 0.457i

 matrix columns 4 thru 4
        -0.148 - 0.457i
> eigenvalues2 = eig(a).["val"]
 eigenvalues2 =
 matrix columns 1 thru 3
              0.554 + 0i               2.15 + 0i         -0.148 + 0.457i

 matrix columns 4 thru 4
        -0.148 - 0.457i
```

This can be applied to any function that evaluates to a list type.

Global Symbol Table

The RLAB symbol table is just a list. It can be referenced with the special symbol `$$` . The symbol-table can be used like other lists, with certain exceptions.

- The global symbol table cannot be copied.
- The global symbol table cannot be destroyed.

Why would you want to use `$$` ? Well, you might use it to reference a variable with a string. For example:

```
> printf ("Enter variable name to display: "); a= getline("stdin");
Enter variable name to display: eps
> $$.[a.[1]]
 eps =
 1.11e-16
```

Another use might be to list the contents of the variables that are defined.

```
>  for (i in members($$))
    {
      if (class ($$.[i]) != "function")
      {
        $$.[i] ?
      }
    }
 eigenvalues1 =
 matrix columns 1 thru 3
             0.554 + 0i                2.15 + 0i          -0.148 + 0.457i

 matrix columns 4 thru 4
        -0.148 - 0.457i
 eigenvalues2 =
 matrix columns 1 thru 3
             0.554 + 0i                2.15 + 0i          -0.148 + 0.457i

 matrix columns 4 thru 4
        -0.148 - 0.457i
 eps =
 1.11e-16
 pi =
     3.14
```

Chapter 7

User-defined Functions

It is assumed that the reader has had some programming experience with a high-level language like Fortran, or even better, a lower-level language like C.

Functions

You have already seen some of the functions in use. There are for basic types of functions that can operate on matricies in RLAB. They are:

Scalar Functions: These functions operate on scalars, and treat matrices in an element-by-element fashion. Some examples are:

abs
cos
sqrt
isnan
exp
sin
real
int
floor
tan
imag
round
ceil
conj

Vector Functions: These functions operate on vectors, either row-vectors (1-by-n) or column vectors (m-by-1), in the same fashion. If the argument is a matrix with both dimensions > 1 then the operation is performed in a column-wise fashion. Some examples are:

sum
min
prod
fft
mean
sort
max
any

When using a vector oriented function, like `max ()`, on a general matrix, it is possible to obtain scalar quantities. For example the maximum value in a matrix can be obtained using `max (max (a))`. The first call to `max ()` returns a vector of the maximum values from each column, and the second call to `max ()` returns the maximum value in the matrix.

Matrix Functions: These functions operate on matrices as a single entity. Some examples are:

balance
hess
pinv
reshape
chol
inv
qr
solve
det
lu
rank
svd
eig
norm
rcond
symm

Functions are Variables

Like matrices, and lists functions are stored as ordinary variables in the symbol table. And, like other variables in the symbol table, functions are accessible as global variables. Function's treatment as variables explains the somewhat peculiar syntax required to create and store a function.

```
> logsin = function ( x ) { return log (x) .* sin (x); }
```

The above statement creates a function, and assigns it to the variable `logsin`. The function can then be used like:

```
> logsin ( 2 )
     0.63
```

Like variables function can be copied, re-assigned, and destroyed.

```
> // Create a function
> logsin = function ( x ) { return log (x) .* sin (x); }
>
> // Use it
> logsin (2)
     0.63
>
> // Copy it to the variable y
> y = logsin
> y (2)
     0.63
>
> // Overwrite it with a matrix
> logsin = rand(3,2)
 logsin =
 matrix columns 1 thru 2
        1      0.333
    0.975     0.0369
    0.647      0.162
>
> // Check that y still is a function
> y (2)
     0.63
```

If you try re-assigning a built-in function you will get a run-time error message. The built-in functions, those that are programmed in C, are a permanent part of the environment. So that users may always rely on their availability, they cannot be re-assigned, or copied.

Variables that represent functions can also be part of list objects. Sometimes it can be useful to group functions that serve a similar purpose, or perform different parts of a larger procedure.

```
list = << logsin = logsin >>
   logsin
> list.logsin (2)
     0.63
> list.expsin = function ( x ) { return exp (x) .* sin (x); }
   expsin       logsin
> list.expsin (2)
     6.72
```

Function Syntax

The function syntax is fairly simple. The basic form of a function is:

function (*argument list*)
{
    local (*local variable list*)

    statements

    return *expression*
}

The `local`, and `return` statements are optional.

If a syntax error is encountered while the function is being entered (read), definition of the function must begin again, from the very beginning.

Local Statement

The local statement is optional. If it is present it must be the first statement in the function. Only one local statement is allowed.

Return Statement

The return statement is optional. There are no restrictions on the number of return statements, or their placement. A function can return from any point in it's execution.

Function Scoping Rules

RLAB's scoping rules will seem somewhat peculiar if you don't remember that: "everything is a variable".

By default, all variables used inside a function, with the exception of arguments, and local variables, are global. This allows users to have access to the other user and builtin functions without special declarations, or complicated scoping rules.

Variables in a function are resolved by:

1. Looking in the fuction arguments,
2. Looking in the local variables,
3. Looking in the file's static variables,
4. Looking in the symbol-table.

Function Arguments

Arguments are passed by reference. Thus, when a function argument is modified, the object in the caller's scope gets modified directly.

A function can be called with fewer arguments than specified in the definition. When this situation occurs, rlab pads the argument list with extra undefined variables. These undefined arguments can be detected with the `exist` function. Sometimes functions are written to use "default" values for arguments that are unspecified. For example: in `ode78.r` the argument variable `tol` is set to the "default" value of `1.e-6` if `tol` is `UNDEFINED`. Note that an undefined variable does not exist. Therefore, the `exist` function will return `0` if it's argument is undefined. There are two methods for invoking a function and specifiying undefined arguments. Either the arguments can be unspecified and the argument list will be padded with undefined variables, or the argument(s) can be explicitly specified as undefined by using a variable that is undefined. The variable `UNDEF` is commonly used for this, but the choice of undefined variable name(s) is arbitrary.

A function cannot be called with more arguments than specified in the function definition. If you attempt to do so, a run-time error will result.

Function argument types are not specified during definition. When writing "robust" functions the author should take some care to check that the function argument(s) are of the correct type. Furthermore, the documentation (comments) should clearly state the requisite argument types if necessary. If the function documentation does not clearly state that the arguments will be modified during function execution, care should be exercised to avoid changing any of the function arguments. If necessary, the function arguments can be copied to local variables, so that changes will not affect the caller's variables.

Function Local Variables

Local variables are created each time the function is invoked. Each local variable is initially `UNDEFINED`. When execution has left the function, the local variables are destroyed.

If you wish to write function(s) to serve as often used utilities or libraries, then care should be taken to declare all variables (other than function arguments) as local. Declaring all function variables as local will prevent accidental destruction of user's global variables.

Function Recursion

Function can call themselves recursively. Each time execution passes into the function the local variables are (re)created. There is a special keyword: `$self`, which can be used to force a function to refer to itself. This is only necessary if you plan to rename the function after it has been created.

```
fac = function ( a )
{
  if(a <= 1)
  {
      return 1;
  else
      return a*fac (a-1);   // return a*$self (a-1);
  }
};
```

In the previous example a factorial computation is performed using recursion[1]. In the second return statement, the function calls itself until $a \leq 1$. In the event that the function is later copied to another variable, and the original destroyed, the function will no longer work. To avoid this, use the special keyword `$self` in place of the function self-reference.

Examples

Many functions are included with the RLAB source distribution. Functions can be found in the distribution subdirectories `./rlib`, `./toolbox`, and `./contrib`.

We will discuss a few examples from the RLAB source distribution.

Mean Example

The rfile `mean.r` (See Figure 7.1) has several noteworthy attributes.

---

[1] Not necessarily an efficient way to compute the factorial

```
//-------------------------------------------------------------------//
//
//  Syntax:     mean ( A )

//  Description:

//  Calculate the mean value. If the input is a 1xN, then compute the
//  mean value of all the elements.

//  If the input is a MxN matrix the compute a row matrix of the mean
//  value of each column of the input.
//
//-------------------------------------------------------------------//

mean = function(x)
{
  local(m);

  m = size (x)[1];
  if( m == 1 )
  {
    m = size (x)[2];
  }

  return sum( x ) / m;
};
```

Figure 7.1: Mean Function

1. The top lines of the file contain comments which document the usage of the function. This is useful, since the the rlab help function will copy the contents of each rfile to the screen. If documentation comments are include in the topmost portion of the file, then users will have convenient access to the help comments.

2. `mean` does not do any error checking on the input arguments. Whether it should or not is debatable. We will present arguments for and against:

   For: Calling `mean` with a string, or a list variable as argument will result in a slightly obscure error message.

   ```
   > mean ( "string constant" )
   rlab: NULL, invalid type for sum()
   near line 25, file: /usr/local/lib/rlab/rlib/mean.r
   > mean ( << [1,2,3]; 100 >> )
   rlab: NULL, invalid type for sum()
   near line 25, file: /usr/local/lib/rlab/rlib/mean.r
   >
   ```

   Instead of `mean` reporting the error, the error is propagated down to `sum`. This is somewhat confusing, since the user committed the error with the function `mean`.

   Against: The function is obviously intended to calculate the mean value of a vector, or matrix object. And, for any numeric argument `mean` will work just fine. Users who are foolish enough to try and compute the mean value of a string, or a heterogeneous object (a list) should not be catered to. That is, users who use the function correctly, should not have to pay a performance penalty.

MGS Example

This example (See Figure 7.2) implements a modified Gram-Schmidt algorithm to find an orthonormal basis for the input matrix. There are several important points to recognize in this function.

1. The function returns a list. `mgs` returns a list containing the matrices Q and R. The members of the list can be accessed by:

```
> aa = rand(4,4)
 aa =
 matrix columns 1 thru 4
      0.7       0.96      0.924      0.148
     0.95      0.915     0.0882      0.879
   0.0918      0.441      0.908    0.00543
    0.902     0.0735      0.362      0.222
> x = mgs (aa)
   q           r
> x.q
 q =
 matrix columns 1 thru 4
     0.47      0.513      0.261     -0.669
    0.638      0.243     -0.613      0.396
   0.0617      0.436      0.642      0.628
    0.606     -0.698      0.379     0.0378
> mgs (aa).q
 q =
 matrix columns 1 thru 4
     0.47      0.513      0.261     -0.669
    0.638      0.243     -0.613      0.396
```

```
//
// Modified Gram-Schmidt
// Given A (MxN), with rank(A) = N. The following algorithm computes
// the factorization A = Q*R (skinny QR) where Q (MxN) has orthonormal
// columns and R (NxN) is upper triangular
//
// From MATRIX Computations, G.H. Golub, C.F. Van Loan (page 219)
//

mgs = function(A)
{
  local(a,k,j,n,m,q,r);

  a = A;
  m = a.nr;
  n = a.nc;
  for(k in 1:n)
  {
    r[k;k] = norm( a[1:m;k], "2");
    q[1:m;k] = a[1:m;k]/r[k;k];
    for(j in k+1:n)
    {
      r[k;j] = q[1:m;k]' * a[1:m;j];
      a[1:m;j] = a[1:m;j] - q[1:m;k] * r[k;j];
    }
   }
  return << q = q; r = r >>;
};
```

Figure 7.2: Modified Gram-Schmidt Function

```
      0.0617       0.436       0.642       0.628
       0.606      -0.698       0.379      0.0378
```

2. The function argument is copied. The argument `A` is copied to the local variable `a` to avoid destroying the original contents of `A` (or `aa` in the callers environment).

Files

Simple "one-liner" functions can be typed in at the command line. However, they are destroyed when the RLAB session is ended. When writing longer functions, or functions that you want to save for repeated usage; it is convenient to create them using a text editor and save them on disk as an ordinary ASCII text file.

The function `load` will execute the rlab statements in a file as if they were typed at the command line. The rlab command `rfile` searches a specified path for files with a '.r' extension. When the rfile command finds a file that matches it's argument, it executes the rlab statements in the file as if they were typed at the command line.

Statements in a file are executed in the same manner as they would be had they been typed in interactively, ordinary commands and multiple functions are O.K. In fact, complete programs can be written and run interactively or in batch mode. To run a program in batch mode you can try:

```
$ rlab program.r &
```

Or the program could contain `#!/usr/local/bin/rlab` on the first line. Then, if your operating system provides the proper support, rlab can execute your program, interactively, or in the background by simply typing:

```
$ chmod +x program.r
$ ./program.r
```

File Static Variables

Although static variables are not peculiar to functions, they are discussed here because they are important when writing packages. A package is a file that contains some combination of statements and/or functions that perform some specific purpose, or provides a specific function.

When writing a package for general use, it is important that the elements of the package do not adversely affect the user's workspace. The user's workspace can be avoided through careful use of the local, and static declarations.

The static declaration restricts the visibility, and accessibility of variables to the file that the static declaration occurs within. Static variables cannot be altered by functions or statements that are not within the same file scope.

Consider the following example: Figure 7.3 contains two functions, and a static declaration. The entire contents of Figure 7.3 is contained within a file named `lu.r`. Since the static declaration occurs within the file `lu.r`, only statements within that file can use the statically declared variable, which, in this case, is the variable swap (a user function). `swap` is used like any other variable, except that it is invisible to any statement outside of the file `lu.r`.

Conclusion

This concludes our simple function tutorial. We have just barely covered some of the capabilities of user-functions. Topics we have not covered, that you may wish to experiment with are:

1. Passing functions as arguments to other functions.
2. Using list variables to get the effect of variable length argument lists.
3. Using list variables, and user-functions to build your own "objects".

```
static (swap);

lu = function ( A )
{
  local (i, l, u, pvt, x)

  if (A.nr != A.nc) { error ("lu() requires square A"); }

  x = factor (A);       // Do the factorization

  //
  // Now create l, u, and pvt from lu and pvt.
  //

  l = tril (x.lu, -1) + eye (size (x.lu));
  u = triu (x.lu);
  pvt = eye (size (x.lu));

  //
  // Now re-arange the columns of pvt
  //

  for (i in 1:max (size (x.lu)))
  {
    pvt = pvt[ ; swap (1:pvt.nc, i, x.pvt[i]) ];
  }
  return << l = l; u = u; pvt = pvt >>;
}

//
//  In vector V, swap elements I, J
//
swap = function ( V, I, J )
{
  local (v, tmp);
  v = V;
  tmp = v[I];
  v[I] = v[J];
  v[J] = tmp;
  return v;
};
```

Figure 7.3: User-Function

Chapter 8

Plotting

RLAB can plot using two methods. The first is to use the plplot library. The other is to use the freely available gnuplot plotting program. The way your RLAB was built determines whether you can use neither, just one, or both methods. gnuplot can produce output on many different display types and file formats, though the quality of the plotting varies according to the capabilities of the output device. plplot does not have the same range of output options, however it has much better support under RLAB .

Plot Process

Setting up Plots

You set the right type of output using `pstart`.

The need to redirect output is less common, though it may occur if you want to import your plots into another package, which requires you to dump the plot to a file.

Setting the output to a file and the terminal type to produce PostScript, and then resetting it to the default setting, is another of the more common things you need to do. You should be careful not to over-write one plot with the next.

Note that PostScript is not the only printer format that gnuplot can produce. Most common printers are either supported directly, or can be configured to emulate a printer that is supported.

Another thing to note is that some terminal types (generally those for printers) take extra options that specify which mode of the terminal should be used. As mentioned above, a complete list of all terminal types (output formats), and the options they take, can be found by invoking gnuplot at the command line, then using the `set terminal` command.

Basic Plots

plplot is a very powerful library, and has many options for specifying how the plots turn out. However, in most of the plots you will do, you will simply be looking for the information contained within the plot, and the defaults will be quite acceptable. The advanced options will be covered in the next section.

The basic command to get information from RLAB to gnuplot is the `plot` command. See Section 9 for detailed reference information. `plot` normally takes either a single matrix, a list of matricies, or a string as its argument. If you wish to, you may also add a second argument, which is the plot process number (see above).

If the argument is a string, the string (without the surrounding double quotes) is passed directly to gnuplot. This was used above, and will be used again when we see the advanced plotting features.

If the argument is a single matrix, then first column is taken as the independant variable, and the second and higher columns are plotted against it.

If the matrix only has one column, then the index values are used as the independant variable, and the column vector is plotted as the dependent variable.

The third type of input is using a list of matricies. This is a very flexible tool, though it has little point unless the matricies have *different* first columns. In this case, the two matricies will be plotted on the same graph, over the domains specified by the first column.

If you want to have two different plots at the same time, you need to specify a plot process number. Here is an example of how to produce three graphs on a system capable of displaying several windows, such as X windows:

The last command shows how to plot multiple curves on a single plot without using a list — remember that each column after the first is treated as a dependant variable to the first column.

The effect of this is best seen by actually trying it on a system capable of displaying multiple plots.

After you have produced multiple plots, you may wish to clean up the screen and free up some resources, so you need to kill some of the plots. You could always exit RLAB, but that is not always desirable. So the `pclose` function is provided. Section 9 has all the details, so suffice to say that you can kill off a plot process by specifying its plot process number as an argument to `pclose`.

With all those plot process numbers floating around, it is easy to lose track of how each one is set up, and which files it is using (or would use).

Advanced Plots

This section deals more with the features of plplot and less with the actual RLAB functions that call those features.

replot If you want to change the settings, and then see the result of plotting a graph again, the `replot` command may be of use. It simply causes the last plot to be redone.

Chapter 9

Function Reference

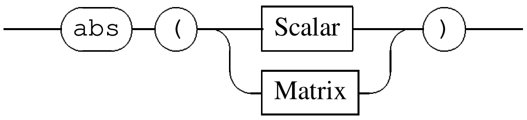
RLAB has a variety of functions that operate on scalars, matricies, strings and lists. There are two types of RLAB functions — built-in and user-defined. Because some user-defined functions are very useful, they are loaded automatically when you start RLAB. These are referred to as standard R-files, and are just files containing useful RLAB code that are stored in a special library directory. In addition, there are other useful functions supplied with the RLAB distribution, and you can also define other functions.

Both built-in functions and standard R-files are documented in this section.

Layout All the functions are layed out in the same general way. The title tells you the name of the function, and what it does. The railroad diagrams tell you what you need to enter. The description tells you what the function returns, and how it works. The example is a short sample showing how you might use this in an RLAB program. Examples are usually pretty simple to keep them short. The See also section gives cross-references to functions that have related functionality or are often used with this function.

abs — Absolute Value

AbsoluteValue



`abs` returns the absolute value of the argument. It operates element-by-element for a matrix argument, returning a matrix result. Absolute value is defined as

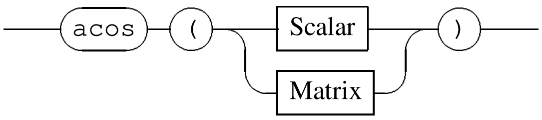
$$\begin{cases} -x & \text{if } x < 0 \text{ and } x \text{ is real} \\ x & \text{if } x \geq 0 \text{ and } x \text{ is real} \\ \sqrt{\Re(x)^2 + \Im(x)^2} & \text{if } x \text{ is complex} \end{cases}$$

Example

```
> abs(2.34)
     2.34
> abs(-2.34)
     2.34
> abs(-2.34 + 4j)
     4.63
> sqrt((-2.34)^2 + (4)^2)
     4.63
> b = [ -3.4, 4.6097; 3+4j , -34.5653 -0.01j]
 b =
               -3.4 + 0i                4.61 + 0i
                  3 + 4i            -34.6 - 0.01i
> abs(b)
      3.4       4.61
        5       34.6
```

acos—Arc Cosine

ArcCosine



`acos` returns the arc cosine of the argument. The return value is expressed in radians, in the range $[0, \pi]$. It is an error if the magnitude of either the real or complex part of the argument exceeds 1. If the argument is a matrix, then the operation is performed element by element.

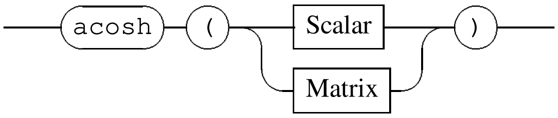
Example

```
> acos(1)
      0
> acos(0)
   1.57
> acos(-1)
   3.14
> acos(0.707+ 0.707j)
  0.999 - 0.764i
> c=0+1j
 c =
      0 + 1i
> acos(c)
   1.57 - 0.881i
> B = [ 0, -0.5 ; -0.3+1j, 0.4-0.003j]
 B =
                  0 + 0i               -0.5 + 0i
               -0.3 + 1i            0.4 - 0.003i
> acos(B)
               1.57 + 0i               2.09 + 0i
           1.78 - 0.897i         1.16 + 0.00327i
```

See also: Page 50 `acosh()`, Page 53 `asin()`, Page 55 `atan()`, Page 56 `atan2()`, Page 72 `cos()`.

acosh—Hyperbolic Arc Cosine

ACosh



`acosh` returns the hyperbolic arc cosine of the argument. The return value is expressed in radians. If the argument is a matrix, the operation is performed element-by-element.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `acosh.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `acosh.r` may make this function unavailable.

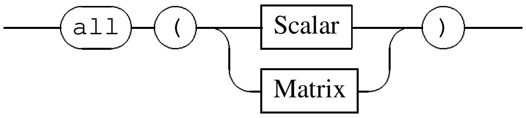
Example

```
> acosh(0)
        0 + 1.57i
> acosh(1+1j)
     1.06 + 0.905i
> acosh(1)
        0
> Mat = (rand(2,3)+rand(2,3)*(0+1j))*5
 Mat =
 matrix columns 1 thru 3
              5 + 3.32i           3.24 + 1.02i          0.185 + 3.27i
          4.87 + 0.423i          1.67 + 0.837i         0.809 + 0.644i

> acosh(Mat)
 matrix columns 1 thru 3
          2.48 + 0.593i            1.9 + 0.319i            1.9 + 1.52i
         2.27 + 0.0885i          1.28 + 0.531i         0.751 + 0.897i
```

See also: Page 49 `acos()`, Page 54 `asinh()`, Page 57 `atanh()`, Page 73 `cosh()`, Page 200 `sinh()`, Page 220 `tanh()`.

all—Test for non-zero matrix

All



`all` tests for non-zero arguments. If the argument is a scalar, the result is 0 if the argument is 0 - otherwise the result is 1. If the argument is a row or column vector, `all` returns a 1 if all of the elements are non-zero. If the argument is some other kind of matrix, `all` returns a row vector produced by performing the test on each column in turn.

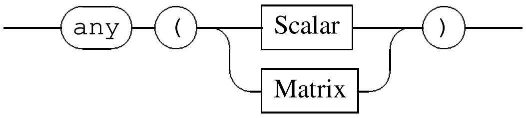
Example

```
> a = [ inf(),3,4,4,0]
 a =
 Infinity          3          4          4          0
> all(a)
        0
> b = [0,0,0]
 b =
        0          0          0
> all(b)
        0
> c = [ nan(),3,4,4,20]
 c =
      NaN          3          4          4         20
> all(c)
        1
> d = [a;b,7,3;c]
 d =
 Infinity          3          4          4          0
        0          0          0          7          3
      NaN          3          4          4         20
> all(d)
        0          0          0          1          0
> all(all(d))
        0
```

See also: Page 52 `any()`.

any — Test for non-zero matrix

Any



`any` tests for non-zero arguments. If the argument is a scalar, the result is 0 if the argument is 0 - otherwise the result is 1. If the argument is a row or column vector, `any` returns a 1 if any of the elements are non-zero. If the argument is some other kind of matrix, `any` returns a row vector produced by performing the test on each column of the matrix in turn.

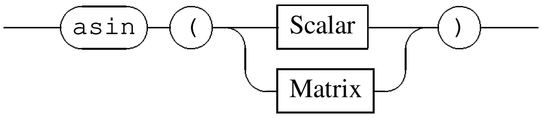
Example

```
> a = [ 0,3,4,4,0]
 a =
        0          3          4          4          0
> any(a)
        1
> b = [0,0,0]
 b =
        0          0          0
> any(b)
        0
> c = [a;b,6,0]
 c =
        0          3          4          4          0
        0          0          0          6          0
> any(c)
        0          1          1          1          0
> any(any(c))
        1
```

See also: Page 51 `all()`.

asin — Arc Sine

ArcSine



`asin` returns the arc sine of the argument. The return value is expressed in radians, in the range $[-\pi/2, \pi/2]$. It is an error if the argument is real and the magnitude of the argument is greater than 1. If the argument is a matrix, the operation is performed element by element.

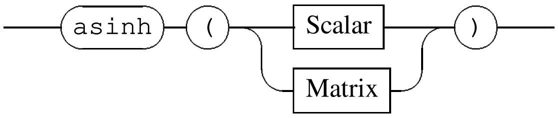
Example

```
> asin(1)
     1.57
> asin(0)
        0
> asin(-1)
    -1.57
> asin(0.5)
    0.524
> asin(0+0.5j)
        0 + 0.481i
> asin(0.5+0.5j)
    0.452 + 0.531i
> Mat = (rand(2,3)+rand(2,3)*(0+1j))
 Mat =
 matrix columns 1 thru 3
              1 + 0.665i          0.647 + 0.204i         0.0369 + 0.655i
         0.975 + 0.0847i          0.333 + 0.167i          0.162 + 0.129i

> asin(Mat)
 matrix columns 1 thru 3
          0.806 + 0.853i          0.677 + 0.259i         0.0309 + 0.616i
           1.23 + 0.254i          0.334 + 0.176i           0.161 + 0.13i
```

See also: Page 49 `acos()`, Page 54 `asinh()`, Page 55 `atan()`, Page 56 `atan2()`, Page 199 `sin()`.

asinh — Hyperbolic Arc Sine

ASinh



`asinh` returns the hyperbolic arc sine of the argument. The return value is expressed in radians. If the argument is a matrix, the operation is performed element-by-element.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `asinh.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `asinh.r` may make this function unavailable.

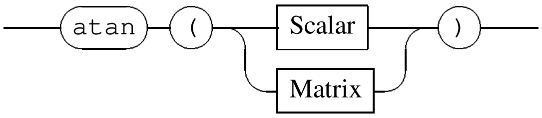
Example

```
> asinh(0)
        0
> asinh(1)
    0.881
> asinh(100)
      5.3
> asinh(0.5+0.5j)
    0.531 + 0.452i
> Mat= (rand(2,3)+rand(2,3)*(0+1j))*5
 Mat =
 matrix columns 1 thru 3
               5 + 3.32i            3.24 + 1.02i           0.185 + 3.27i
           4.87 + 0.423i           1.67 + 0.837i          0.809 + 0.644i

> asinh(Mat)
 matrix columns 1 thru 3
            2.49 + 0.58i           1.93 + 0.294i            1.86 + 1.51i
          2.29 + 0.0849i           1.36 + 0.415i          0.823 + 0.494i
```

See also: Page 50 `acosh()`, Page 53 `asin()`, Page 57 `atanh()`, Page 73 `cosh()`, Page 200 `sinh()`, Page 220 `tanh()`.

atan — Arc Tangent

ArcTangent



`atan` returns the arc tangent of the argument. The return value is expressed in radians, in the range $[-\pi/2, \pi/2]$. If the argument is a matrix the operation is performed element-by-element.

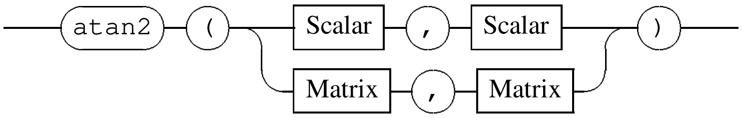
Example

```
> atan(0)
        0
> atan(1)
    0.785
> pi/2
     1.57
> atan(-inf())
    -1.57
> atan(inf())
     1.57
> atan(1+1j)
     1.02 + 0.402i
> Mat= (rand(2,3)+rand(2,3)*(0+1j))*5
 Mat =
 matrix columns 1 thru 3
               5 + 3.32i            3.24 + 1.02i          0.185 + 3.27i
           4.87 + 0.423i           1.67 + 0.837i         0.809 + 0.644i

> atan(Mat)
 matrix columns 1 thru 3
          1.43 + 0.0907i          1.29 + 0.0822i          1.55 + 0.314i
           1.37 + 0.017i            1.1 + 0.197i         0.807 + 0.365i
```

See also: Page 49 `acos()`, Page 53 `asin()`, Page 56 `atan2()`, Page 57 `atanh()`, Page 219 `tan()`.

atan2 — Arc Tangent of Ratio

ArcTan2



`atan2` returns the arc tangent of the result of dividing the first argument by the second argument. Both arguments must be real. This should be at least as accurate as, and probably faster than, performing the division then using `atan`. The result is expressed in radians, in the range $[-\pi, \pi]$. If the arguments are matricies, then they must have the same dimensions, and the operation is performed element-by-element.

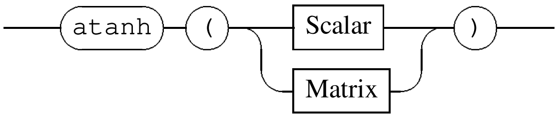
Example

```
> atan2(1,2)
    0.464
> atan2(0,2)
        0
> atan2(2,0)
     1.57
> atan2(0,0)
        0
> b = rand(2,5)
 b =
        1      0.647     0.0369      0.665      0.204
    0.975      0.333      0.162     0.0847      0.167
> c = rand(2,5)
 c =
    0.655       0.91      0.299        0.7     0.0918
    0.129      0.112      0.265       0.95      0.902
> b./c
     1.53      0.711      0.124       0.95       2.22
     7.57       2.98      0.609     0.0891      0.185
> atan2(b,c)
    0.991      0.618      0.123       0.76       1.15
     1.44       1.25      0.547     0.0889      0.183
```

See also: Page 49 `acos()`, Page 53 `asin()`, Page 55 `atan()`, Page 57 `atanh()`, Page 219 `tan()`.

atanh — Hyperbolic Arc Tangent

ATanh



`asinh` returns the hyperbolic arc tangent of the argument. The return value is expressed in radians. If the argument is a matrix, the operation is performed element-by-element.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `atanh.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `atanh.r` may make this function unavailable.

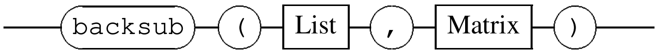
Example

```
> atanh(0)
        0
> atanh(pi/2)
    0.752 + 1.57i
> atanh(pi)
     0.33 + 1.57i
> atanh(2+2j)
    0.239 + 1.31i
> d = rand(2,5)
 d =
     0.96      0.441      0.924      0.908      0.148
    0.915     0.0735     0.0882      0.362      0.879
> atanh(d)
     1.94      0.474       1.62       1.52      0.149
     1.56     0.0737     0.0884      0.379       1.37
```

See also: Page 50 `acosh()`, Page 54 `asinh()`, Page 55 `atan()`, Page 73 `cosh()`, Page 200 `sinh()`, Page 220 `tanh()`.

backsub — solution of linear equations

backsub



The `backsub` function computes the solution to the set of linear equations described by:

$$AX = B$$

The list argument to `backsub` is the result from `factor(A)`. The second argument to `backsub` is the matrix B, where each column of B is a seperate right hand side.

`backsub` returns X, the solution of the linear equations. Each column of X corresponds to that column of B.

Example

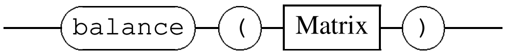
```
> A = rand(4,4)
 A =
    0.493      0.161      0.137      0.879
    0.782     0.0642      0.446      0.761
    0.591      0.271      0.367      0.852
   0.0721     0.0384      0.949      0.107
> X1 = rand(4,1)
 X1 =
    0.812
    0.543
    0.254
   0.0825
> B1 = A*X1
 B1 =
    0.595
    0.846
    0.791
    0.329
> X2 = rand(4,1)
 X2 =
    0.697
    0.137
    0.815
    0.974
> B2 = A*X2
 B2 =
     1.33
     1.66
     1.58
    0.933
> B = [B1,B2]
 B =
    0.595       1.33
    0.846       1.66
    0.791       1.58
    0.329      0.933
> F = factor(A)
```

```
   lu            pvt           rcond
> backsub(F,B)
    0.812      0.697
    0.543      0.137
    0.254      0.815
   0.0825      0.974
> // This is [X1,X2] - so it works!
```

See also: Page 92 `factor()`, Page 119 `inv()`, Page 203 `solve()`.

balance — Matrix Balancing

Balance



`balance` takes a square argument matrix and attempts to balance the input matrix so that the row and column norms are approximately equal. It returns a list with elements `t` and `Ab`, such that for an argument A:

$$Ab = t^{-1}At$$

Ab is the balanced matrix, and t is referred to as the transformation matrix.

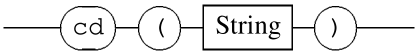
Example

```
> A = [ 1, 10, 100; 2,-20,200; -100, 3, -30]
 A =
        1         10        100
        2        -20        200
     -100          3        -30
> norm(A)
      330
> norm(A')
      222
> res = balance(A)
 res =
   ab            t
> res.ab
 Ab =
        1        100        100
      0.2        -20         20
     -100         30        -30
> norm(res.ab)
      150
> norm(res.ab')
      201
```

See also: Page 142 `norm()`.

cd — Change Directory

CD



`cd` changes the location of the current working for RLAB. This affects any search paths that contain **.** (full stop), representing the current directory. It also changes the directory in which the diary files are written, and the directory in which the file read and write functions work on. The argument string supplies an absolute or relative pathname which specifies the new directory.

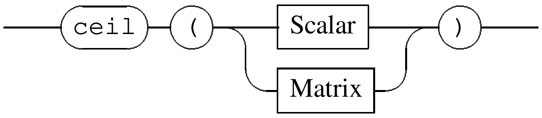
Example

```
> system("pwd")
/home/bradh
        0
> cd("rlab")
        1
> system("pwd")
/home/bradh/rlab
        0
> cd("../archived")
        1
> system("pwd")
/home/bradh/archived
        0
> // This is a sym-link on my machine
> cd("/linux")
        1
> system("pwd")
/usr/src/linux
        0
```

See also: Page 218 `system()`.

ceil — Ceiling Value

Ceil



`ceil` returns the smallest integer not less than the argument. If the argument is a matrix then the `ceil` operation is performed on an element-by-element basis.

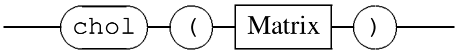
Example

```
> ceil(4.56)
        5
> ceil(4.999999999)
        5
> ceil(4.000000001)
        5
> a=100*rand(2,5)
 a =
     65.5          91        29.9          70        9.18
     12.9        11.2        26.5          95        90.2
> ceil(a)
       66          92          30          70          10
       13          12          27          96          91
```

See also: Page 100 `floor()`, Page 116 `int()`, Page 189 `round()`.

chol — Cholesky Factorisation

Cholesky



`chol` computes the Cholesky factorization of the argument matrix. The input matrix must be real symmetric positive definite, or complex Hermitian positive definite. `chol` returns an upper triangular matrix U, such that U^TU and the argument are equal.

Example

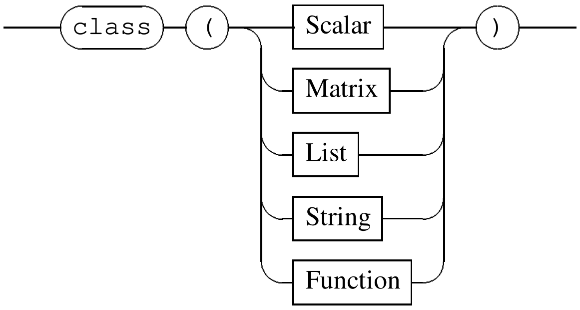
```
> b = rand(4,4)
 b =
    0.941      0.383      0.103      0.381
    0.121      0.448      0.821      0.871
    0.402      0.801      0.715      0.555
    0.712      0.705      0.311      0.385
> a = b*b’
 a =
     1.19      0.702       0.97       1.12
    0.702       1.65       1.48      0.993
     0.97       1.48       1.62       1.29
     1.12      0.993       1.29       1.25
> u = chol(a)
 u =
     1.09      0.644       0.89       1.03
        0       1.11      0.814      0.299
        0          0      0.408      0.319
        0          0          0     0.0686
> u’*u
     1.19      0.702       0.97       1.12
    0.702       1.65       1.48      0.993
     0.97       1.48       1.62       1.29
     1.12      0.993       1.29       1.25
> c = rand(3,3)+rand(3,3)*1j;
> b = c’*c
 b =
 matrix columns 1 thru 3
               1.67 + 0i           0.983 + 0.41i           1.27 + 0.467i
           0.983 - 0.41i               1.05 + 0i           0.56 - 0.155i
           1.27 - 0.467i           0.56 + 0.155i               1.49 + 0i

> u = chol(b)
 u =
 matrix columns 1 thru 3
               1.29 + 0i          0.761 + 0.317i          0.982 + 0.361i
                  0 + 0i              0.612 + 0i         -0.492 - 0.193i
                  0 + 0i                  0 + 0i              0.345 + 0i

> u’*u
 matrix columns 1 thru 3
               1.67 + 0i           0.983 + 0.41i           1.27 + 0.467i
```

```
     0.983 - 0.41i             1.05 + 0i           0.56 - 0.155i
      1.27 - 0.467i          0.56 + 0.155i              1.49 + 0i
```

See also: Page 130 `lu()`.

class — Type of entity

Class



`class` returns a string which identifies what the argument is. There are four possible return strings

`num` for scalar and numeric matricies.

`string` for strings and string matricies

`list` for lists

`function` for built-in and user defined functions.

This function is very useful inside user-defined functions, for both error checking and decision making.

The class of a variable or function can also be determined by using the class member reference, as shown below. This technique does *not* work for a list unless you have given it a `class` element.

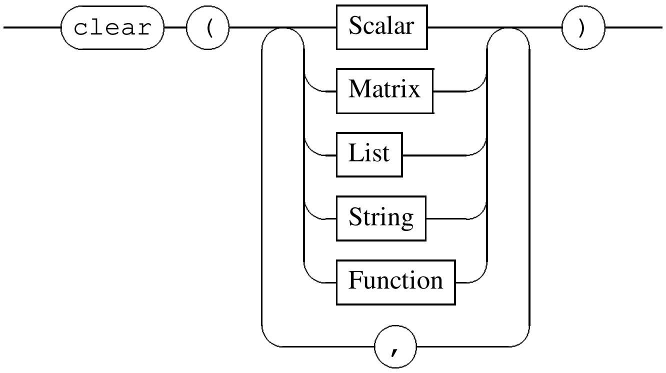
Example

```
> class(3)
num
> class([3,-3,1;2,6,3])
num
> class("any 'ol string")
string
> class(["any 'ol string","and another","to make a string matrix"])
string
> class(<<a = 3;b = "string">>)
list
> class(sin)
function
> class(sin(3))
num
> "any 'ol string".class
string
> g = 4+3j
 g =
        4 + 3i
> g.class
num
> sin.class
function
```

See also: Page 195 `show()`, Page 229 `what()`.

clear — Clear variable or function

Clear



`clear` is a function that allows you to clean up the symbol table. It deletes the variables and functions that you specify in the argument list. `clear` accepts up to 32 arguments, and returns the number of objects that have been successfully cleared. You can not `clear` a built-in function. You should beware of clearing the values of `pi` and `eps`, as many R-files make implicit use of these.

If you wish to clear the whole symbol table, use the `clearall` function.

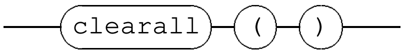
Example

```
> a = 0.647
 a =
    0.647
> b = rand(2,2)
 b =
   0.0847      0.167
    0.204      0.655
> string3 = "Jus' another string"
Jus' another string
> F = << a;b>>
   1            2
> who()
F        b        pi
a        eps      string3
> clear(F, pi)
        2
> who()
a        b        eps      string3
> clear(a,b,string3)
        3
> who()
eps
> a = cumsum([1,2,3,4,5])
 a =
        1          3          6          10          15
> clear(cumsum);
> b = cumsum([1,2,3,4,5])
rlab: cumsum, UNDEFINED
```

See also: Page 67 `clearall()`, Page 231 `who()`.

clearall — Erase all variables

ClearAll



`clearall` clears all data objects from the workspace. Scalars, strings, matrices, and lists are cleared with the `clear` function. As a special exception, `eps` and `pi` are not affected. User function are also not affected. If you wish to remove user functions you must do so explicitly with clear.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `clearall.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `clearall.r` may make this function unavailable.

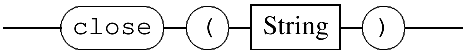
Example

```
> a = rand(3,3);
> b = "a string";
> c = 3 + 4j;
> who()
a    b    c    eps  pi
> clearall();
> who()
eps  pi
```

See also: Page 66 `clear()`.

close — Close a File

Close



`close` takes a single string argument, and tries to close the file named by the string. It returns 1, or true, if the file was closed. It returns 0, or false, if the file could not be closed.

You have to close a file between writing to it, and reading from it.

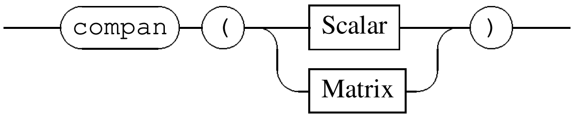
Example

```
> close("No such File")
        0
> write("tmp_file_name",rand(2,2))
        1
> close("tmp_file_name")
        1
```

See also: Page 182 `read()`, Page 233 `write()`.

compan — Companion matrix

Companion



`compan` returns the companion matrix of the argument matrix. If the argument is an (n+1)-vector, the companion matrix is nxn. If the argument is a scalar, say s, then `compan(s)` is the sxs matrix formed by taking the companion of $[1, 2, \ldots, s + 1]$. If the argument is a matrix, but is not a vector, an internal conversion to vector format is done.

$\Longrightarrow$ This is not an RLAB built-in function. This function is normally loaded on start-up from the `compan.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `compan.r` may make this function unavailable.

Example

```
> compan(1)
         1
> compan(2)
        -2          -3
         1           0
> compan(3)
        -2          -3          -4
         1           0           0
         0           1           0
> compan([1,2])
         1
> compan([1,2,3])
        -2          -3
         1           0
> compan([2,3,5])
      -1.5        -2.5
         1           0
```

complement — Complement of a set

Complement

`complement` takes two sets (either row or column vectors), and finds the complement of the first set in the second set. This means that it returns the elements of the second set that are not in the first set.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `complement.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `complement.r` may make this function unavailable.

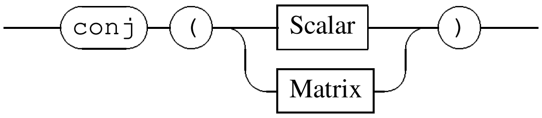
Example

```
> a = [1,4,-45,24,4.65]
 a =
        1          4        -45          24       4.65
> b = [0,433,43,1,4.65,4,2,24]
 b =
 matrix columns 1 thru 6
        0        433         43           1       4.65              4

 matrix columns 7 thru 8
        2         24
> complement(a,b)
        0          2         43        433
```

See also: Page 118 `intersection()`, Page 194 `set()`, Page 228 `union()`.

conj — Complex Conjugate

Conj



`conj` returns the complex conjugate of its argument. For matrix arguments the conjugation is performed element by element.

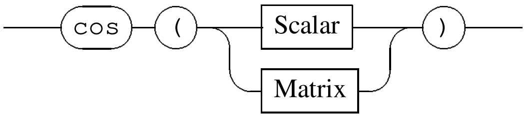
Example

```
> conj(2)
        2
> conj(2+5.762j)
        2 - 5.76i
> conj(2-5.762j)
        2 + 5.76i
> a = (100*(rand(2,3)+rand(2,3)*(0+1j)))-(50+50j)
 a =
 matrix columns 1 thru 3
              50 + 16.5i            14.7 - 29.6i           -46.3 + 15.5i
            47.5 - 41.5i           -16.7 - 33.3i           -33.8 - 37.1i

> conj(a)
 matrix columns 1 thru 3
              50 - 16.5i            14.7 + 29.6i           -46.3 - 15.5i
            47.5 + 41.5i           -16.7 + 33.3i           -33.8 + 37.1i
```

See also: Page 113 `imag()`, Page 185 `real()`.

cos — Cosine

Cosine



`cos` calculates the cosine of the argument, which is taken as being specified in radians. If the argument is a matrix, the `cos` operation is performed element by element. If the argument is complex, given by z, then the result is:

$$\cos(\Re(z))\cosh(\Im(z)) - j\sin(\Re(z))\sinh(\Im(z))$$

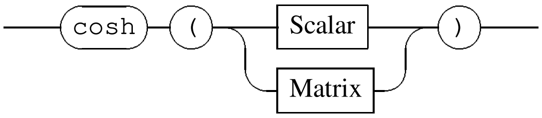
Example

```
> cos(0)
        1
> cos(pi/2)
 6.12e-17
> cos(pi)
       -1
> cos(pi/4)
    0.707
> cos(1+1j)
    0.834 - 0.989i
> a = rand(2,5)*pi
 a =
     3.14       2.03      0.116       2.09      0.641
     3.06       1.05      0.508      0.266      0.526
> cos(a)
       -1     -0.447      0.993     -0.494      0.801
   -0.997      0.501      0.874      0.965      0.865
```

See also: Page 49 `acos()`, Page 73 `cosh()`, Page 199 `sin()`, Page 219 `tan()`.

cosh — Hyperbolic Cosine

Cosh



`cosh` calculates the hyperbolic cosine of the argument, specified in radians. If the argument is a matrix, then the operation is performed element-by-element.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `cosh.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `cosh.r` may make this function unavailable.

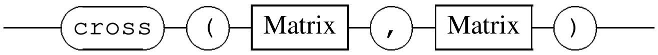
Example

```
> cosh(0)
        1
> cosh(pi)
     11.6
> cosh(0+1j)
     0.54
> cosh(0+10j)
   -0.839
> cosh(pi/2)
     2.51
> a = rand(2,5)*4
 a =
        4        2.59       0.148        2.66       0.817
      3.9        1.33       0.647       0.339       0.669
> cosh(a)
     27.3         6.7        1.01        7.17        1.35
     24.7        2.03        1.22        1.06        1.23
```

See also: Page 50 `acosh()`, Page 54 `asinh()`, Page 57 `atanh()`, Page 72 `cos()`, Page 90 `exp()`, Page 200 `sinh()`, Page 220 `tanh()`.

cross — Vector cross product

Cross



`cross` calculates the vector cross product of the two argument matrices. Both arguments must have three elements, though they need not be organised the same way. The result of the cross product is returned as a $1x3$ matrix.

$\Longrightarrow$ This is not an RLAB built-in function. This function is normally loaded on start-up from the `cross.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `cross.r` may make this function unavailable.

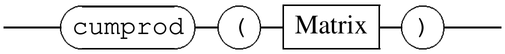
Example

```
> a = [0,0,1]
 a =
        0          0          1
> b = [1;0;0]
 b =
        1
        0
        0
> c = cross(a,b)
 c =
        0          1          0
> d = cross(a,c)
 d =
       -1         -0          0
> d = cross(c,a)
 d =
        1         -0          0
> e = [1,1,1]
 e =
        1          1          1
> cross(a,e)
       -1          1          0
> cross(c,e)
        1         -0         -1
```

See also: Page 82 `dot()`.

cumprod — Cumulative Product

CumProd



`cumprod` takes a matrix argument, and produces another matrix of the same dimensions. Each element in the resultant matrix consists of the cumulative product of the that element and all previous elements in that column of the argument matrix. If the argument is either a row or column vector, the cumulative product is performed on that row or column.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `cumprod.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `cumprod.r` may make this function unavailable.

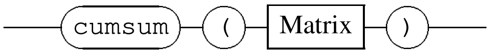
Example

```
> cumprod(1:5)
        1          2          6         24        120
> cumprod([1,2,3,4]')
        1
        2
        6
       24
> c = [1,2,3; 4,5,6; 10, -3.4, 2.09]
 c =
        1          2          3
        4          5          6
       10       -3.4       2.09
> cumprod(c)
        1          2          3
        4         10         18
       40        -34       37.6
> d = [2.3+4j, 5-4j,3.2654+34.5j]
 d =
 matrix columns 1 thru 3
                2.3 + 4i                  5 - 4i            3.27 + 34.5i

> cumprod(d)
 matrix columns 1 thru 3
                2.3 + 4i            27.5 + 10.8i             -283 + 984i
```

See also: Page 76 `cumsum()`.

cumsum — Cumulative Summation

CumSum



`cumsum` takes a matrix argument, and produces another matrix of the same dimensions. Each element in the resultant matrix consists of the cumulative sum of the that element and all previous elements in that column of the argument matrix. If the argument is either a row or column vector, the cumulative summation is performed on that row or column.

$\Longrightarrow$ This is not an RLAB built-in function. This function is normally loaded on start-up from the `cumsum.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `cumsum.r` may make this function unavailable.

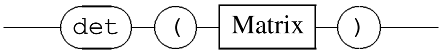
Example

```
> cumsum([1,2,3,4,5])
        1          3          6         10         15
> cumsum([1,2,3,4]')
        1
        3
        6
       10
> c = [1,2,3; 4,5,6; 7,8,9; 10, -3.4, 2.09]
 c =
        1          2          3
        4          5          6
        7          8          9
       10       -3.4       2.09
> cumsum(c)
        1          2          3
        5          7          9
       12         15         18
       22       11.6       20.1
```

See also: Page 75 `cumprod()`.

det — Determinant

Determinant



`det` calculates the determinant of the matrix argument. The argument must be square and non-singular.

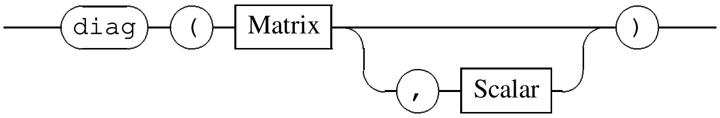
Example

```
> det([2,3; 4,5])
       -2
> b = [2,4;-4,-8]
 b =
        2          4
       -4         -8
> det(b)
rlab: b, matrix is singular
> c = rand(3,3)
 c =
        1      0.333      0.665
    0.975     0.0369     0.0847
    0.647      0.162      0.204
> c = rand(4,4)
 c =
    0.167      0.112       0.95      0.915
    0.655      0.299     0.0918      0.441
    0.129      0.265      0.902     0.0735
     0.91        0.7       0.96      0.924
> det(c)
   0.0765
> d = rand(3,3)+rand(3,3)*(0+1j)
 d =
 matrix columns 1 thru 3
          0.414 + 0.278i           0.29 + 0.737i          0.791 + 0.363i
          0.233 + 0.789i          0.205 + 0.248i          0.269 + 0.369i
          0.555 + 0.692i           0.561 + 0.45i          0.334 + 0.505i

> det(d)
   -0.152 + 0.242i
```

See also: Page 119 `inv()`.

diag — Diagonalise matrix

Diagonalise



`diag` has two forms, depending on what type of matrix the first argument is. If the first argument is a row or column vector, then `diag` returns a square matrix with that vector on a diagonal.

Which diagonal the vector is placed on is determined by the second argument, which defaults to 0 if you don't specify it. The main diagonal is taken as 0. The diagonal above the main diagonal are taken as positive integers, and the diagonals below the main diagonal are negative integers.

The second form of `diag` is used if the first argument is not a row or column vector. The result of this operation is a column matrix consisting of all the elements taken from a diagonal. As for the first form, you can use the optional scalar argument to specify which diagonal to extract the elements from. The notation is the same, as is the default of 0.

As an implementation curiosity, if you specify a matrix as the second argument, then it uses the first element of that matrix to determine the diagonal to use.

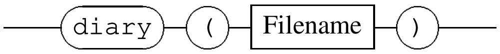
Example

```
> a = rand(3,5)
 a =
        1      0.333      0.665      0.167       0.91
    0.975     0.0369     0.0847      0.655      0.112
    0.647      0.162      0.204      0.129      0.299
> diag(a)
        1
   0.0369
    0.204
> diag(a,0)
        1
   0.0369
    0.204
> diag(a,-1)
    0.975
    0.162
> diag(a,2)
    0.665
    0.655
    0.299
> b = 1:3
 b =
        1          2          3
> diag(b)
        1          0          0
        0          2          0
        0          0          3
> diag(b,1)
        0          1          0          0
        0          0          2          0
        0          0          0          3
        0          0          0          0
```

See also: Page 225 `tril()`, Page 226 `triu()`.

diary — Log File

Diary



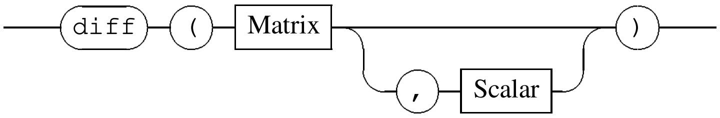
The `diary` function writes a log of whatever happens in the current RLaB session after the `diary` function is called. Both user input and program responses are written to the file specified by the string argument, or to the default diary file `./DIARY`, if no argument is supplied. When the file is opened, a string containing the name of the diary file and the date and time are written out to the diary file. Invoking `diary` a second time with no arguments will close the currently opened diary file. At most one diary file may be in use at any one time.

Example

```
> diary()
        1
> a = 2
 a =
        2
> b = 3+rand()*1j
 b =
        3 + 1i
> diary()
        1
> system("more DIARY");
RLaB diary file: DIARY. Opened Wed Apr 27 09:36:15 1994

        1
> a = 2
 a =
        2
> b = 3+rand()*1j
 b =
        3 + 1i
> diary()
> diary("DiaryFile23");
> a = rand(2,3)+rand(2,3)*1j
 a =
 matrix columns 1 thru 3
         0.975 + 0.0847i          0.333 + 0.167i           0.162 + 0.129i
          0.647 + 0.204i         0.0369 + 0.655i            0.665 + 0.91i

> diary();
> system("more DiaryFile23");
RLaB diary file: DiaryFile23. Opened Wed Apr 27 09:42:54 1994

> a = rand(2,3)+rand(2,3)*1j
 a =
 matrix columns 1 thru 3
         0.975 + 0.0847i          0.333 + 0.167i           0.162 + 0.129i
          0.647 + 0.204i         0.0369 + 0.655i            0.665 + 0.91i

> diary();
```

diff — Difference between matrix elements

Difference



`diff` calculates the difference between elements of the matrix argument. If the matrix is a vector, then `diff` returns a vector of the differences between adjacent elements. The return vector has one less element than the argument. If the argument is not a vector, then differences are calulated down each column, returning a matrix with one less row than the argument.

If the optional scalar argument is present, it specifies the number of times differences are taken. For example, if it is 2, then the differences between the differences are returned.

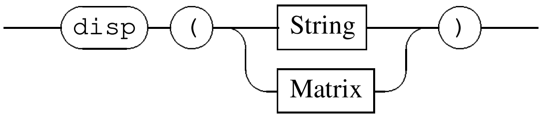
⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `diff.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `diff.r` may make this function unavailable.

Example

```
> a = [1,2,3,4,5]
 a =
        1          2          3          4          5
> diff(a)
        1          1          1          1
> b = [1,2,4,7,11]
 b =
        1          2          4          7         11
> diff(b)
        1          2          3          4
> diff(diff(b))
        1          1          1
> diff(b,2)
        1          1          1
> diff(b,3)
        0          0
> c = [1,1,1;
>      2,2,2;
>      3,4,5;
>      4,7,11]
 c =
        1          1          1
        2          2          2
        3          4          5
        4          7         11
> diff(c)
        1          1          1
        1          2          3
        1          3          6
> diff(c,2)
        0          1          2
        0          1          3
> diff(c,3)
        0          0          1
```

disp — Display entity

Disp



`disp` is a useful function for writing out entities. If the argument is a matrix, then it is printed, without the variable label, to standard output. If the argument is a string, it prints the string to standard output. The return value from this function is 1 if the function succeeded, otherwise it is zero.

⟹ This is not an RLaB built-in function. This function is normally loaded on start-up from the `disp.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `disp.r` may make this function unavailable.

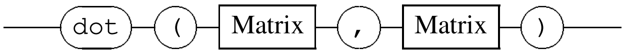
Example

```
> a = rand(2,4)
 a =
        1      0.647     0.0369      0.665
    0.975      0.333      0.162     0.0847
> disp(a)
        1      0.647     0.0369      0.665
    0.975      0.333      0.162     0.0847
        1
> disp(a);
        1      0.647     0.0369      0.665
    0.975      0.333      0.162     0.0847
> b = "any string will do"
any string will do
> disp(b)
any string will do
        1
> disp(b);
any string will do
```

See also: Page 170 `printf()`, Page 172 `printmat()`.

dot — Vector dot product

Dot



`dot()` calculates the dot product of two argument vectors. The vectors should have the same number of elements. The resultant is a scalar consisting of the sum of the element by element product of the two vectors.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `dot.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `dot.r` may make this function unavailable.

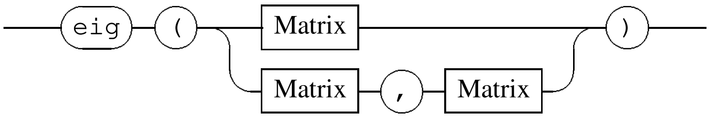
Example

```
> a = 1:5
a =
1          2          3          4          5
> b = 6:2:-1
b =
6          5          4          3          2
> a.*b
6         10         12         12         10
> 6+10+12+12+10
50
> dot(a,b)
50
> c = b';
> dot(a,c)
       50
```

See also: Page 74 `cross()`+, Page 213 `sum()`, Page 224 `trace()`.

eig — Eigen Decomposition

Eigen



`eig` with a single square argument matrix computes that matrix's eigenvectors and eigenvalues. The results are returned as a list with elements `val` and `vec` which are the eigenvalues and right eigenvectors respectively. In general, these will be complex quantities.

`eig` with two arguments computes the eigenvectors and eigenvalues of the real generalized symmetric (or complex generalized Hermitian) definite eigenproblem. The eigenvalues and eigenvectors are returned in a list as for the first form. If the two arguments are A and B, the eigenproblem is defined by

$$Ax = \lambda Bx$$

Both forms of `eig` check for symmetry, and use an appropriate solver.

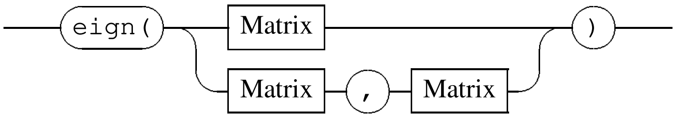
Example

```
> A = rand(3,3)
 A =
    0.108      0.741     0.0646
    0.297      0.329     0.0287
    0.254      0.345      0.499
> eigen= eig(A)
 eigen =
   val          vec
> A* eigen.vec[;1] - eigen.val[1]*eigen.vec[;1]
           1.39e-16 + 0i
          -1.11e-16 + 0i
           4.86e-17 + 0i
> A* eigen.vec[;2] - eigen.val[2]*eigen.vec[;2]
           1.11e-16 + 0i
          -3.33e-16 + 0i
          -1.11e-16 + 0i
> A* eigen.vec[;3] - eigen.val[3]*eigen.vec[;3]
           9.71e-17 + 0i
          -1.39e-16 + 0i
          -1.11e-16 + 0i
```

See also: Page 84 `eign()`, Page 85 `eigs()`.

eign — Non-symmetric Eigen Decomposition

EigenN



`eign` with a single square argument matrix computes that matrix's eigenvectors and eigenvalues. The results are returned as a list with elements `lvec`, `val` and `rvec` which are the left eigenvectors, eigenvalues and right eigenvectors respectively. In general, these will be complex quantities.

`eign` with two arguments computes the eigenvectors and eigenvalues of the real generalized symmetric (or complex generalized Hermitian) definite eigenproblem. The eigenvalues and eigenvectors are returned in a list as for the first form.

This routine forces use of the non-symmetric eigensolver — no checking is done.

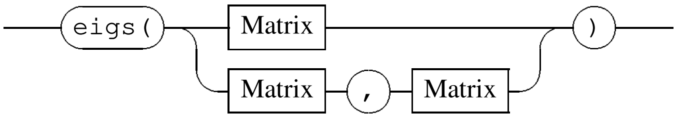
Example

```
> D = rand(4,4)
 D =
    0.512      0.522      0.111      0.693
    0.807      0.253      0.216      0.762
    0.397      0.339     0.0222      0.717
    0.361     0.0251      0.356     0.0606
> issymm(D)
        0
> eigenN = eign(D)
 eigenN =
   lvec         rvec         val
> D * eigenN.rvec[;1] - eigenN.val[1]* eigenN.rvec[;1]
           3.33e-16 + 0i
                  0 + 0i
                  0 + 0i
                  0 + 0i
> D * eigenN.rvec[;2] - eigenN.val[2]* eigenN.rvec[;2]
          -2.29e-16 + 0i
          -1.39e-16 + 0i
          -1.11e-16 + 0i
          -9.71e-17 + 0i
> D * eigenN.rvec[;3] - eigenN.val[3]* eigenN.rvec[;3]
           1.94e-16 + 0i
           6.94e-17 + 0i
           5.55e-17 + 0i
          -5.55e-17 + 0i
```

See also: Page 83 `eig()`, Page 85 `eigs()`.

eigs — Symmetric Eigen Decomposition

EigenS



`eigs` with a single square argument matrix computes that matrix's eigenvectors and eigenvalues. The results are returned as a list with elements `val` and `vec` which are the eigenvalues and right eigenvectors respectively. In general, these will be complex quantities.

`eigs` with two arguments computes the eigenvectors and eigenvalues of the real generalized symmetric (or complex generalized Hermitian) definite eigenproblem. The eigenvalues and eigenvectors are returned in a list as for the first form. If the two arguments are A and B, the eigenproblem is defined by

$$Ax = \lambda Bx$$

This routine forces use of the symmetric eigensolver — no checking is done.

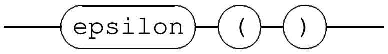
Example

```
> B = rand(3,3)
 B =
     0.57      0.831      0.462
    0.683     0.0735      0.435
    0.396      0.464       0.63
> C = symm(B)
 C =
     0.57      0.757      0.429
    0.757     0.0735       0.45
    0.429       0.45       0.63
> eigenS = eigs(C)
 eigenS =
   val          vec
> C* eigenS.vec[;1] - eigenS.val[1] * eigenS.vec[;1]
 2.78e-16
 5.55e-17
 2.22e-16
> C* eigenS.vec[;2] - eigenS.val[2] * eigenS.vec[;2]
-1.39e-16
-6.25e-17
-1.67e-16
> C* eigenS.vec[;3] - eigenS.val[3] * eigenS.vec[;3]
-4.44e-16
 2.22e-16
        0
```

See also: Page 83 `eig()`, Page 84 `eign()`.

epsilon — Compute machine epsilon

Epsilon



`epsilon` calculates the epsilon of this machine. Epsilon is defined as the largest number you can add to 1 and get a result that is indiscernable from 1. There is a variable, `eps`, that is automatically placed into the global symbol table by the default `.rlab` file. You may find the variable more convenient to use than the function, and the expense of possible portablity problems.

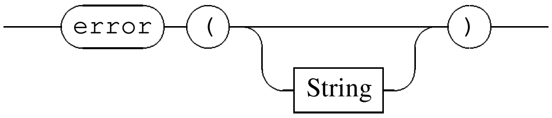
⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `epsilon.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `epsilon.r` may make this function unavailable.

Example

```
> epsilon()
 1.11e-16
> eps
 eps =
 1.11e-16
> 1+epsilon()==1
        1
> 1+2*epsilon()==1
        0
```

error — Raise an error

Error



The `error` function allows user defined functions to jump back to the prompt when they detect an error has occurred. The nature of the error message displayed is up to the user. If no argument is supplied, `error` will print the default message, which is `USER-RAISED-ERROR`. If a string arguments is supplied, then that string is displayed.

Jumping back to the prompt means execution of the current loop or function is terminated immediately, and the control is returned to the RLAB command line.

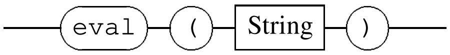
Example

```
> error()
rlab: USER-RAISED-ERROR
> error("The argument was of the wrong form")
rlab: The argument was of the wrong form
> error("Have you got any idea what you are doing?")
rlab: Have you got any idea what you are doing?
```

See also: Page 170 `printf()`.

eval — Evaluate expression

Eval



`eval` is a way to make up commands and functions at execution time. You do not gain much by using this function during an interactive session, however it can be very useful when called from a script.

The expression contained in the string argument is evaluated without regard to local functions. Essentially what occurs is equivalent to replacing the `eval` function with the contents of the string in a *global* context. Any assignments or variable modifications will effect the global symbol table only.

`eval` does not have any knowledge of local variables. Thus if you use `eval` within a user-defined function, you must be careful not to reference that function's arguments or local variables.

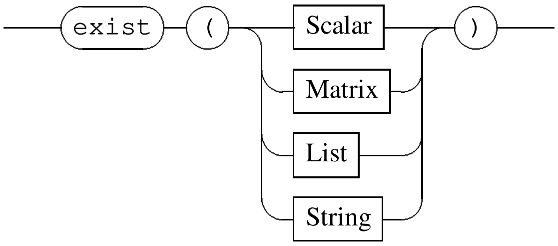
⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `eval.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `eval.r` may make this function unavailable.

Example

```
> eval("a = cos(0)")
 a =
       1
       0
> eval("a = sin(sqrt(2)/2)");
 a =
    0.65
> eval("b = tan(3);")
       0
> b
 b =
  -0.143
> printf( "Enter a function: " ); x = getline( "stdin" );
Enter a function: sqrt
> // this could be in a function
> eval("tmp = " + x.[1] + "(" + "3" + ")" );
 tmp =
    1.73
> printf( "Enter a function: " ); x = getline( "stdin" );
Enter a function: tanh
> eval("tmp = " + x.[1] + "(" + "3" + ")" );
 tmp =
   0.995
```

exist — Test for an argument

Exist



`exist` is a function that tests whether its argument is actually a legitimate variable or function. If the argument exists, then `exist` returns 1, or true if you prefer, and returns 0, or false, if the argument doesn't exist.

Note that you can't pass a constant value as an argument, and passing an argument that has enough brackets to look like a function, but isn't really a function, gives an error message. This is caused by RLAB not being able to evaluate the function.

In general, you should be using this function in general purpose user-defined functions to make sure all the neccessary arguments are present, and to provide defaults as appropriate.

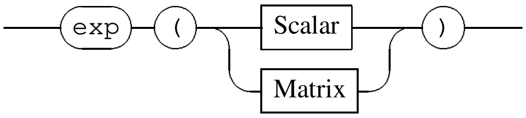
Example

```
> who()
eps  pi
> exist(eps)
        1
> exist(sin)
        1
> exist(sin(2))
        1
> exist(2)
rlab: invalid argument to exist,
```

See also: Page 229 `what()`, Page 231 `who()`.

exp — Exponential

Exp



`exp` returns e raised to the power of the argument. If the argument is a matrix, then an element-by-element operation is performed. If the argument is complex value, z , then the result is:

$$e^{\Re(z)}\cos(\Im(z)) + je^{\Re(z)}\sin(\Im(z))$$

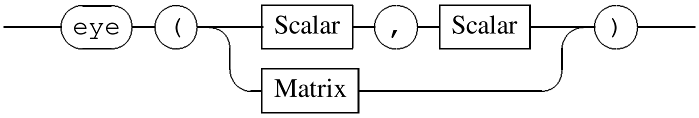
example

```
> exp(0)
        1
> exp(1)
     2.72
> exp(2)
     7.39
> exp(1)^2
     7.39
> exp(54)
 2.83e+23
> exp(3.2)
     24.5
> exp(-23)
 1.03e-10
> exp([0,1,2,5,34,2.543])
 matrix columns 1 thru 6
        1       2.72       7.39        148   5.83e+14       12.7
```

See also: Page 127 `log()`, Page 128 `log10()`.

eye — Identity matrix

Identity



`eye` generates an identity matrix. This is a matrix with each element on the main diagonal set to 1 and all other elements set to 0. If the arguments are two scalars, the first specifies the number of rows and the second specifies the number of columns in the resultant. If the argument is a matrix, then it must have two elements, with the first element specifying the number of rows, and the second element specifying the number of columns in the result.

⟹ This is not an RLaB built-in function. This function is normally loaded on start-up from the `eye.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `eye.r` may make this function unavailable.

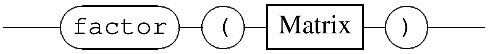
Example

```
> eye(3,3)
        1          0          0
        0          1          0
        0          0          1
> eye(2,5)
        1          0          0          0          0
        0          1          0          0          0
> eye([3,4])
        1          0          0          0
        0          1          0          0
        0          0          1          0
> c = rand(2,4)
 c =
    0.846      0.826       0.29      0.438
    0.683      0.252      0.494      0.324
> eye(size(c))
        1          0          0          0
        0          1          0          0
```

See also: Page 78 `diag()`, Page 146 `ones()`, Page 239 `zeros()`.

factor — LU Factorisation

Factor



`factor` computes the LU factorization of the square, non-singular argument matrix. It returns a list with 3 elements:

`lu` a matrix containing the LU factors

`pvt` a vector containing the pivot indices

`rcond` the reciprocal of the condition estimate

`factor` returns the results in the above format, so that they may be conveniently used with `backsub` for repetitive solutions. The `lu` function uses the results from `factor` to produce separate L and U matrices.

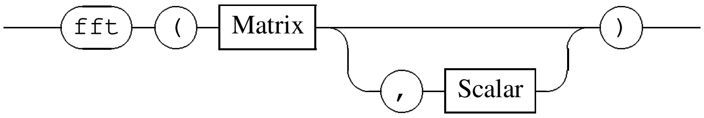
Example

```
> M = rand(4,4)
 M =
        1     0.0369      0.204       0.91
    0.975      0.162      0.167      0.112
    0.647      0.665      0.655      0.299
    0.333     0.0847      0.129      0.265
> a = factor(M)
   lu           pvt          rcond
> a.lu
        1     0.0369      0.204       0.91
    0.647      0.641      0.523      -0.29
    0.975      0.196     -0.134     -0.718
    0.333      0.113    -0.0132    -0.0144
> a.pvt
        1          3          3          4
> a.rcond
 0.000436
```

See also: Page 58 `backsub()`, Page 119 `inv()`, Page 203 `solve()`.

fft — Fourier Transform

FFT



`fft` calculates the forward Fourier Transform of the argument matrix. If the argument is a row or column vector, then the transformation is performed on that vector. However if the argument is not such a vector, then the transformation is performed on each column in turn.

The result is *not* scaled.

An optional scalar argument can be used to zero-pad or to truncate the argument to a certain length.

To perform two dimensional Fourier Transforms, repeated calls to `fft` are required, taking the second call in the other direction. This is shown below.

Example

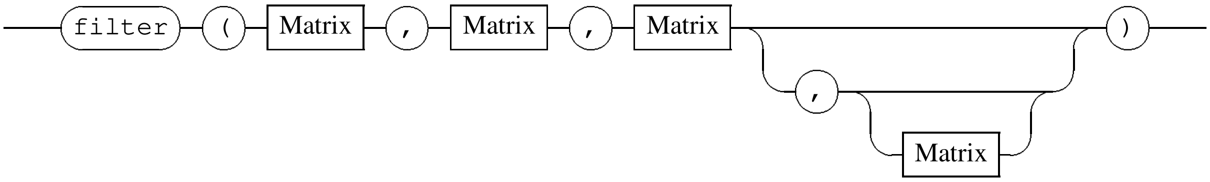
```
> a = 0:2*pi:0.5
 a =
 matrix columns 1 thru 6
        0        0.5          1        1.5            2        2.5

 matrix columns 7 thru 12
        3        3.5          4        4.5            5        5.5

 matrix columns 13 thru 13
        6
> sin(a)
 matrix columns 1 thru 6
        0      0.479      0.841      0.997      0.909      0.598

 matrix columns 7 thru 12
    0.141     -0.351     -0.757     -0.978     -0.959     -0.706

 matrix columns 13 thru 13
   -0.279
> fft(sin(a))
 matrix columns 1 thru 3
            -0.0617 + 0i           0.605 - 6.35i         -0.126 + 0.286i

 matrix columns 4 thru 6
         -0.115 + 0.141i        -0.112 + 0.0816i        -0.111 + 0.0439i

 matrix columns 7 thru 9
        -0.111 + 0.0139i        -0.111 - 0.0139i        -0.111 - 0.0439i

 matrix columns 10 thru 12
        -0.112 - 0.0816i         -0.115 - 0.141i         -0.126 - 0.286i

 matrix columns 13 thru 13
           0.605 + 6.35i
> a = rand(4,4)
 a =
```

```
        1     0.0369      0.204       0.91
    0.975      0.162      0.167      0.112
    0.647      0.665      0.655      0.299
    0.333     0.0847      0.129      0.265
> fft(fft(a)')
 matrix columns 1 thru 3
             6.64 + -0i         -0.115 + 0.603i               2.19 + -0i
           1.8 + 0.638i            1.03 + 1.84i          -0.223 + 0.377i
              1.58 + 0i        -0.0817 + 0.757i              -0.384 + 0i
           1.8 - 0.638i          0.573 - 0.636i          -0.223 - 0.377i

 matrix columns 4 thru 4
        -0.115 - 0.603i
         0.573 + 0.636i
       -0.0817 - 0.757i
           1.03 - 1.84i
> abs(fft(fft(a)'))
     6.64      0.614       2.19      0.614
     1.91       2.11      0.438      0.856
     1.58      0.761      0.384      0.761
     1.91      0.856      0.438       2.11
```

See also: Page 112 `ifft()`.

filter — Digital Filter Structure

Filter



`filter` is a framework that allows you to performing filtering using finite impulse response (FIR) or infinite impulse response (IIR) techniques.

Each matrix argument must be a real row or column vector (or a scalar, which will be converted to a one element vector). The first argument contains the zeros of the transfer function — call it $\hat{b}$, containing nb terms from $[b_1, b_2, \ldots, b_{nb}]$. The next argument contains the poles of the transfer function — call it $\hat{a}$, containing na terms from $[a_1, a_2, \ldots, a_{na}]$. [1].

Note that the a_1 term must not be zero, as all the other terms are divided by this term in a preprocessing stage. If you are attempting to make a FIR filter (zeros only), this term is normally set to 1.

The next argument is the input stream to the filter, usually denoted as $\hat{x}$, containing nx terms, from$[x_1, x_2, \ldots, x_{nx}]$.

The last arg is an optional vector of the initial values in each delay bin. It must have number of terms equal to the maximum of the length of A or B.

The filter outputs are in a list with elements named `y`, which is the output of the filter - it has the same number of terms as the input; and `zf`, which is a vector of the final values in each bin.

The result is calculated by

$$y_n = b_1 x_n + b_2 x_{n-1} + \ldots + b_{nb+1} x_{n-nb} - a_2 y_{n-1} - \ldots - a_{na+1} y_{n-na}$$

Example

```
> // We want a bandpass filter, unity gain from 0.2 to 0.4
> // Refer Williams and Taylor, 'Electronic Filter Design Handbook',
> // Example 13.1, page 13-8.  31st order FIR.  Use their co-efficients
> b = [ 0.03820,0.01551,0.08376,-0.12525,-0.05134,0.01481,-0.01461, ...
0.03616,-0.08645,-0.03592,0.00906,0.01462,0.14963,-0.25786,-0.12825, ...
0.41256,-0.12825,-0.25786,0.14963,0.01462,0.00906,-0.03592,-0.08645, ...
0.03616,-0.01461,0.01481,-0.05134,-0.12525,0.08376,0.01551,0.03820]
 b =
 matrix columns 1 thru 6
   0.0382     0.0155     0.0838     -0.125    -0.0513     0.0148

 matrix columns 7 thru 12
  -0.0146     0.0362    -0.0864    -0.0359    0.00906     0.0146

 matrix columns 13 thru 18
     0.15     -0.258     -0.128      0.413     -0.128     -0.258

 matrix columns 19 thru 24
     0.15     0.0146    0.00906    -0.0359    -0.0864     0.0362

 matrix columns 25 thru 30
  -0.0146     0.0148    -0.0513     -0.125     0.0838     0.0155
```

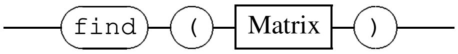
---

[1] Some text books(e.g. Rabiner and Gold, Theory and Application of Digital Signal Processing) use these same variables in the opposite sense

```
 matrix columns 31 thru 31
   0.0382
> a = 1;
> x = [1; zeros(99,1)];
> // this is useful for the plot
> t = 0:1:0.01;
> output = filter(b,a,x);
> plot([t’,log(fft(x))]);
> plot([t’,log(fft(output.y))]);
```

find — Find non-zero elements

Find



`find` finds the indices of the non-zero elements of the argument matrix. The argument is treated as being a vector, so the return values are just element numbers, not row and column numbers. The format of the return values is a row vector.

This function can be used to test for elements that meet a certain criteria.

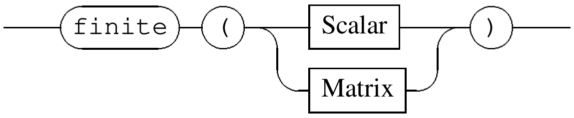
Example

```
> b = eye(3,3)
 b =
        1          0          0
        0          1          0
        0          0          1
> find(b)
        1          5          9
> c = [nan(),3,0,2,0,0,1.43,inf()]
 c =
 matrix columns 1 thru 7
      NaN          3          0          2          0          0       1.43

 matrix columns 8 thru 8
 Infinity
> c = [3,0,2,0,0,1.43,inf()]
 c =
 matrix columns 1 thru 7
        3          0          2          0          0       1.43   Infinity

> find(c)
        1          3          6          7
> d = rand(2,5)
 d =
      0.7     0.0918       0.96      0.441      0.924
     0.95      0.902      0.915     0.0735     0.0882
> d < .5
        0          1          0          1          0
        0          0          0          1          1
> find(d < .5)
        3          7          8         10
```

See also: Page 51 `all()`, Page 52 `any()`.

finite — Test for finite values

Finite



This function tests the argument for values that are not finite. If the argument is scalar, it returns 0 if the argument is finite (i.e. not Inf or NaN) and a 1 if it is either Inf or NaN. If the argument is a matrix, then `finite` returns a matrix of the same dimensions as the argument, with the test performed on each element.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `finite.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `finite.r` may make this function unavailable.

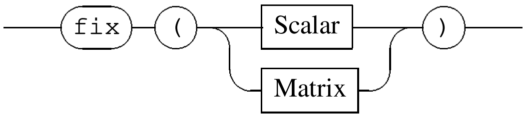
Example

```
> c = inf();
> d = nan();
> e = 3+4j;
> finite(c)
0
> finite(d)
0
> finite(e)
1
> f = [ -c,d,0,2,inf()]
f =
-Infinity        NaN          0           2   Infinity
> finite(f)
        0          0          1           1          0
```

See also: Page 114 `inf()`, Page 121 `isinf()`, Page 122 `isnan()`, Page 141 `nan()`.

fix — Round towards zero

Fix



`fix` rounds the argument towards zero. If the argument is a matrix, this is performed element-by-element.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `fix.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `fix.r` may make this function unavailable.

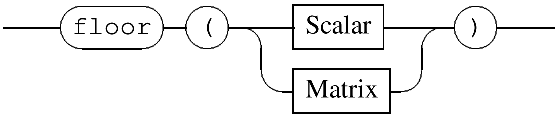
Example

```
> fix(2.53)
        2
> fix(-3.0999)
       -3
> fix(3.9999)
        3
> fix(1.234+4.32j)
        1 + 4i
> fix(1.999+4.999j)
        1 + 4i
> a = rand(2,3)+rand(2,3)*1j;
> b = 10*a
 b =
 matrix columns 1 thru 3
            2.04 + 2.99i               6.55 + 7i            9.1 + 0.918i
            1.67 + 2.65i             1.29 + 9.5i            1.12 + 9.02i

> fix(b)
 matrix columns 1 thru 3
                  2 + 2i                  6 + 6i                  9 + 0i
                  1 + 2i                  1 + 9i                  1 + 9i
```

See also: Page 62 `ceil()`, Page 100 `floor()`, Page 189 `round()`.

floor — Floor Value

Floor



`floor` returns the largest integer not greater than the argument. If the argument is a matrix then the `floor` operation is performed on an element-by-element basis. If the argument is complex, the operation is performed on real and imaginary parts separately.

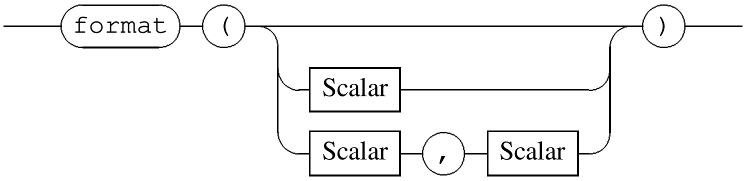
Example

```
> floor(1.00001)
        1
> floor(1.999999)
        1
> floor(-1)
       -1
> floor(-1.73)
       -2
> a = 100*rand(2,5)
 a =
     65.5         91       29.9         70       9.18
     12.9       11.2       26.5         95       90.2
> floor(a)
       65         91         29         69          9
       12         11         26         95         90
> b = 100*rand(2,5)
 b =
       96       44.1       92.4       90.8       14.8
     91.5       7.35       8.82       36.2       87.9
> floor(a+b*(0+1j))
 matrix columns 1 thru 3
                65 + 95i                91 + 44i                29 + 92i
                12 + 91i                 11 + 7i                 26 + 8i

 matrix columns 4 thru 5
                69 + 90i                 9 + 14i
                95 + 36i                90 + 87i
```

See also: Page 62 `ceil()`, Page 116 `int()`.

format — Change output format

Format



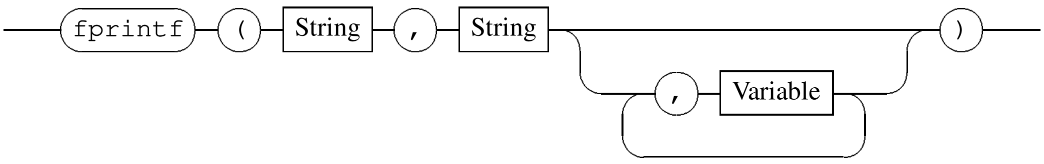
`format` changes the way in which numbers are displayed. If it is invoked with no arguments, then output format is reset to the default values. If you supply a single argument, that is the new precision, which is the number of digits to the right of the decimal place. If you supply two arguments, then the first is the new width, and the second is the new precision. Width is the total number of characters requires, including a decimal point. The default value is a width is nine, and a precision of three.

Example

```
> a = rand(2,3)*100+rand(2,3)*100j;
> a
 a =
 matrix columns 1 thru 3
            38.7 + 98.8i            66.5 + 51.8i            8.59 + 98.9i
            4.92 + 25.5i            8.85 + 21.7i            33.4 + 49.1i

> format(8);
> a
 a =
 matrix columns 1 thru 2
      38.723569 + 98.787275i      66.502073 + 51.786062i
      4.9153286 + 25.491575i      8.8473302 + 21.728892i

 matrix columns 3 thru 3
      8.5900179 + 98.919052i
      33.449489 + 49.135113i
> format(10,5)
         1
> a
 a =
 matrix columns 1 thru 2
          38.724 + 98.787i          66.502 + 51.786i
          4.9153 + 25.492i          8.8473 + 21.729i

 matrix columns 3 thru 3
            8.59 + 98.919i
          33.449 + 49.135i
```

fprintf — Formatted output to a file

FilePrint



The `fprintf` function is intended for writing formatted output to a file. It is similar to the C language function of the same name, though some features are limited or not available, since RLAB doesn't have all the data types of C.

The first argument is a string which specifies which file the formatted string is to be written to. If this argument starts with `|`, then the rest of the argument is taken as a process which should be invoked, and the formatted output string is sent to that process.

The second string argument is the format string. It consists of the text to be written out, and possibly some conversion specifications. A conversion specification is a sequence of commands that determine how the remainder of the arguments are to be displayed. The left most conversion specifier is matched to the third argument, the next specifier to the fourth argument, and so on. Each argument has to be matched to a conversion specification. When the format string is being scanned, the argument that matches the the specifier that the scanner is up to is called the *current argument*.

Conversion specifications always begin with a `%` sign. The next thing that can occur are the flags:

– causing the conversion to be left-justified, instead of the default right-justified.

+ causing a sign to always be prepended to the conversion, instead of the default of only prepending minus signs.

which causes the format to be of an alternate form. This only has meaning for `e`, `E`, `f`, `g` and `G` formats, where is causes a decimal point to be used always. It also prevents suppression of trailing zeros for `g` and `G` options.

0 (zero) which causes padding by leading zeros, instead of the default space padding. This is overridden by both `-` and specifying a precision.

space which causes a space to be prepended if no sign is present. This is overridden by the + flag.

Following any flags, a minimum field width may be specified. This is either a integer constant, or a `*` character. If it is a constant, this is the minimum width. If it is a `*`, then the current argument (which must be a scalar) is used to specify the minimum width. If there is an optional width, then there may also be an option precision. This is specified in the same way as the width, using either and integer constant, or a `*` to signify that the current argument is to be used.

The next thing that can occur is an optional `h` or `l` (ell) modifier. This changes the behavior of the subsequent `i` and `u` conversion specifiers. It is legal, but has no effect, with the `d` specifier. It is illegal with all other specifiers.

The final thing that must occur is a character specifying the way in which the current variable is to be displayed. The valid characters are

c causes the argument to be converted to an unsigned character format.

d which causes the output to be displayed as a decimal string of the form *[-]dddd*. The precision specifies the minimum number of digits to appear, with a default of one.

e causes the output to be displayed in scientific notation of the form *[-]d.ddd*`e`*±dd*, where *d* is any digit. There is always one non-zero digit before the decimal place if the argument is non-zero. The precision specification sets how many digits are present after the decimal place, with a default of six.

E is the same as `e`, except that a upper case letter `E` is used to seperate the mantissa and exponent instead of a lower case `e`.

f causes the output to be displayed as a decimal string of the form *ddd.ddd*. the precision specifies how many digits should appear after the decimal place, with a default of six.

g is the same as `e` is the exponent would be less than -4, or greater than the precision. Otherwise, it is the same as `f`.

G is the same as `E` is the exponent would be less than -4, or greater than the precision. Otherwise, it is the same as `f`.

i is the same as `d`, except that the `l` flag may be used.

s may be used to display strings. If the argument is a scalar, then this is the same as `f`.

u displays the argument as an unsigned integer value. The precision specifies the minimum number of digits to be displayed, with a default value of one.

The number of arguments must match the number of conversion specifications, including those required for width and precision specifications.

`fprintf` cannot print out whole matrices or lists. `write` knows how to deal with entire data objects.

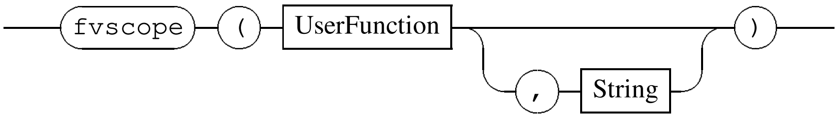
Example

```
Dummy Example
```

See also: Page 170 `printf()`, Page 182 `read()`, Page 206 `sprintf()`, Page 233 `write()`.

fvscope — Scope of a function's arguments

FVScope



The `fvscope` function allows you to determine what variables are used in a function, and what their scope is. The first argument is the name of the function that you wish to examine, and the optional second argument allows you to specify a file to write the resulting analysis to.

`fvscope` is useful for writing general purpose functions that will be used by others. It can be used to identify errant global variables (variables that should be local, but were overlooked).

Note that the line numbers in the first part of the example are all 1. This is a result of entering the function at the command line. If you are using a function from a file, as in the second and third examples, the numbers come out right.

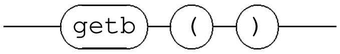
Example

```
> tansum = function (a,b,c)
  {
  local(totsum);
  totsum = a+b+c;
  return(tan(totsum));
  }
        <user-function>
> fvscope(tansum);
        Function Variable SCOPE analysis for : tansum
        Filename: stdin

        line    GLOBAL                  ARG             LOCAL

           1                                            Local-Var: totsum
           1                            Arg-Var: a
           1                            Arg-Var: b
           1                            Arg-Var: c
           1    Global-Var: tan
           1                                            Local-Var: totsum
> fvscope(hilb);
        Function Variable SCOPE analysis for : hilb
        Filename: /usr/local/lib/rlab/rlib/hilb.r

        line    GLOBAL                  ARG             LOCAL

          15                                            Local-Var: i
          15                            Arg-Var: n
          16                                            Local-Var: j
          16                            Arg-Var: n
          17                                            Local-Var: h
          17                                            Local-Var: i
          17                                            Local-Var: j
          17                                            Local-Var: i
          17                                            Local-Var: j
```

```
         20                                               Local-Var: h
> // Note the use of global functions (=variables)
> fvscope(acosh);
        Function Variable SCOPE analysis for : acosh
        Filename: /usr/local/lib/rlab/rlib/acosh.r

        line    GLOBAL                  ARG             LOCAL

          13    Global-Var: log
          13                            Arg-Var: x
          13    Global-Var: sqrt
          13                            Arg-Var: x
```

getb —

GetB



`getb`

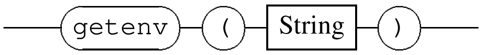
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

getenv — Get Environmental Variable

GetEnv



`getenv` searches the environment list for a string that matches the string argument. The value of the environment variable is returned as a string.

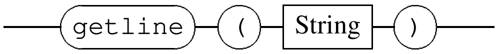
The `getenv` function is mostly a wrapper around the C language function of the same name. This means that its behavior depends upon the underlying implementation. On systems with ISO C libraries, `getenv` will return a zero length string if the environment variable does not exist.

Example

```
> getenv("HOME")
/home/bradh
> getenv("NOT_HOME")

> my_name = getenv("LOGNAME")
 my_name =
bradh
> my_machine = getenv("HOSTTYPE")
 my_machine =
i386
```

getline — Read scalars and text

Getline



The `getline` function is used to read in a line of text from the file specified in the argument, and return that line as a list. Each time you call `getline`, another line is read from the file. To get back to the start of the file, you have to use `close`. `getline` uses spaces and tabs as delimiters of words. The first word on the line becomes the first element in the list, the second word on the line becomes the second element, and so on. `getline` is smart enough to recognise numbers and strings, and can return either or both as elements in the list. Numbers are recognised in normal or in exponential notation. You can get around this by putting the number in `"`-type quotation marks. Numbers are not recognised in complex notation. A list is always returned — if the line was empty, you an empty list is returned.

The argument can also be used to specify a process to run instead of a file to read from. The main limitation is that the process has to be able to write to standard output, which most tools can do. To set this up, just make | the first character of the filename. You can get even more flexibility by using a filter on the original process.

A common operation is to read the whole file. Since `getline` returns an empty list when there is no input, we can tell when to terminate the input loop by checking for a zero length result.

Example

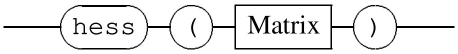
```
> printf("Go ahead, enter some stuff! "); res = getline("stdin");
Go ahead, enter some stuff! two strings "one string" 2+3j 1e99
> res.[1]
 1 =
two
> res.[2]
 2 =
strings
> res.[3]
 3 =
one string
> res.[4]
 4 =
2+3j
> show(res.[4])
   name:      4
   class:     string
    type:     string
     nr:      1
     nc:      1
> res.[5]
 5 =
    1e+99
> show(res.[5])
   name:      5
   class:     num
   type:      real
     nr:      1
     nc:      1
> // a more complex example
> // open the process and throw away the first line
```

```
> getline("|df")
   1            2            3            4            5
   6            7
> tot = 0;
> // The != 0 can be left out, since when length = 0,
> // the if test fails.  In here for clarity
> while (length(result = getline("|df")) != 0)
  {
  tot = tot + result.[2];
  }
> tot
 tot =
 4.58e+05
> system("df");
Filesystem         1024-blocks  Used Available Capacity Mounted on
/dev/hdb5             250559  162300    75732     68%   /
/dev/hda1             207240  175444    31796     85%   /dosC
> 250559+207240
 4.58e+05
```

See also: Page 148 `pause()`.

hess — Hessenberg Matrix

Hessenberg



`hess` finds the Hessenberg form of a matrix. It takes a square matrix, say A, as input, and returns a list with two elements, `h` and `p`, which are the H and P matricies respectively, such that:

$$A = PHP^T$$

This is also known as finding a similar matrix, where A and H are said to be similar, and P is the change of base matrix.

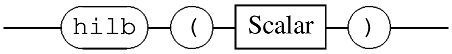
Example

```
> A = rand(3,3)
 A =
     0.965      0.239      0.415
     0.883       0.54      0.625
     0.858     0.0691     0.0172
> L = hess(A)
    h            p
> L.h
 h =
     0.965     -0.461      0.132
     -1.23      0.633    -0.0267
         0       0.53     -0.076
> L.p
 p =
         1          0          0
         0     -0.717     -0.697
         0     -0.697      0.717
> L.p*L.h*L.p'
     0.965      0.239      0.415
     0.883       0.54      0.625
     0.858     0.0691     0.0172
```

See also: Page 193 `schur`.

hilb — Hilbert Matrix

Hilbert



`hilb` generates a Hilbert matrix. The scalar argument specifies the dimensions of the square matrix to be generated. This is also known as the order of the Hilbert matrix. If we have a Hilbert matrix of order n, and row and column indicies of i and j respectively, then the matrix elements are given by

$$a_{ij} = \frac{1}{i + j - 1}$$

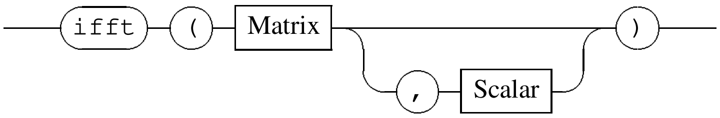
$\Longrightarrow$ This is not an RLAB built-in function. This function is normally loaded on start-up from the `hilb.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `hilb.r` may make this function unavailable.

Example

```
> hilb(1)
        1
> hilb(2)
        1         0.5
      0.5       0.333
> hilb(4)
        1         0.5       0.333        0.25
      0.5       0.333        0.25         0.2
    0.333        0.25         0.2       0.167
     0.25         0.2       0.167       0.143
> rcond(hilb(4))
 3.52e-05
> rcond(hilb(23))
  2.7e-20
```

ifft — Inverse Fourier Tranform

IFFT



`ifft` computes a discrete Fourier transform of the input. Unlike `fft`, the output is scaled by the reciprocal of the number of elements in the input, such that a call to `fft` followed by a call to ifft() will reproduce the original matrix. The resultant is always complex.

The argument must be a matrix. If it is a row or column matrix then a Fourier transformation is performed on that vector. If it is not such a vector, then Fourier transform is performed on each column in turn.

The optional scalar argument can be used to zero-pad or to truncate the argument to the length given as the scalar.

To perform two dimensional Inverse Fourier Transforms, repeated calls to `ifft` are required, taking the second call in the other direction. This is shown below.

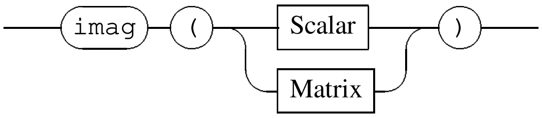
Example

```
> a = rand(2,4)
 a =
        1      0.647     0.0369      0.665
    0.975      0.333      0.162     0.0847
> ifft(a)
 matrix columns 1 thru 3
              0.987 + 0i               0.49 + 0i             0.0993 + 0i
             0.0127 + 0i              0.157 + 0i            -0.0624 + 0i

 matrix columns 4 thru 4
              0.375 + 0i
               0.29 + 0i
> imag(ifft(a))
        0          0          0          0
        0          0          0          0
> b = rand(4,4)
 b =
    0.204       0.91        0.7       0.96
    0.167      0.112       0.95      0.915
    0.655      0.299     0.0918      0.441
    0.129      0.265      0.902     0.0735
> ifft(ifft(b)')'
 matrix columns 1 thru 3
              0.486 + 0i       -0.0931 + 0.0502i            -0.011 + -0i
        0.0804 + 0.0484i       -0.128 - 0.00638i       -0.0608 - 0.0376i
             0.0466 + 0i         0.101 - 0.0262i            -0.109 + -0i
        0.0804 - 0.0484i     -0.00398 - 0.00519i       -0.0608 + 0.0376i

 matrix columns 4 thru 4
       -0.0931 - 0.0502i
     -0.00398 + 0.00519i
         0.101 + 0.0262i
       -0.128 + 0.00638i
```

See also: Page 93 `fft()`.

imag — Imaginary Part

Imag



`imag` returns the imaginary part of the argument. If the argument is a matrix, then the operation is performed element-by-element, returning a matrix.

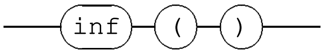
Example

```
> imag(2.4 -3.6j)
     -3.6
> imag(2.4)
        0
> sqrt(-2)
        0 + 1.41i
> imag(sqrt(-2))
     1.41
> b = [2+3j, 4-6j; 3.324, 2-45.6543j]
 b =
                  2 + 3i                  4 - 6i
               3.32 + 0i               2 - 45.7i
> imag(b)
        3         -6
        0      -45.7
```

See also: Page 71 `conj()`, Page 185 `real()`.

inf — Infinity Value

Infinity



`inf` returns a scalar which is set to the IEEE-754 infinity value. What this value can be used for depends on the exact setup of your RLaB implementation.

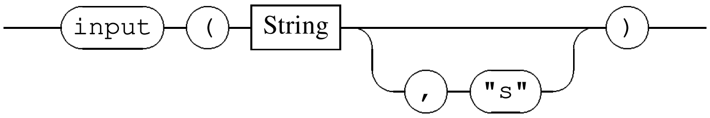
Example

```
> inf()
 Infinity
> a = inf()
 a =
 Infinity
> B = [ 2,3; inf(), -inf()]
 B =
        2          3
 Infinity  -Infinity
```

See also: Page 121 `isinf()`, Page 141 `nan()`.

input — Get user response

Input



The `input` function provides an easy method for users to get a simple response from the keyboard. The string argument is printed on the standard output (usually the screen), and the program waits for the user response. `input` then returns the input, which can be either a string, or a number. If you want to force the input to be a string, then use the optional second argument, `"s"`, to force the return value to be a string. Actually any argument will do — just so long as there is some argument.

If the user just hits return when prompted, then `input` returns an empty matrix.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `input.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `input.r` may make this function unavailable.

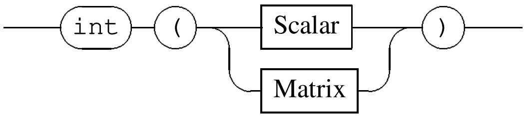
Example

```
> response = input("Don’t just stand there! Do something:");
Don’t just stand there! Do something:text number
> response
 response =
text
> response2 = input("Don’t just stand there! Do something:");
Don’t just stand there! Do something:"sentance with words"
> response2
 response2 =
sentance with words
> Number = input("Gimme a number:")
Gimme a number:3.245
 Number =
     3.25
> show(Number)
   name:      Number
   class:     num
   type:      real
     nr:      1
     nc:      1
> Number = input("Gimme a number:", "s")
Gimme a number:2.3456
 Number =
2.346
> type(Number)
string
```

See also: Page 108 `getline()`.

int — Integer Filter

Int



`int` truncates its argument by conversion to an integer. If the argument is a matrix then the `int` operation is performed on an element-by-element basis.

Because RLaB is not an arbitrary precision system, you may occasionally be surprised by the results of this function, as shown in the example below, where we can 'truncate' out input of 1.9999999999999999 to 2, which is how it is stored internally.

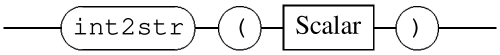
Example

```
> int(1.0001)
        1
> int(1.5)
        1
> int(1.999999999999999)
        1
> int(1.9999999999999999)
        2
> 1.999999999999999
        2
> b = 100*(rand(2,3)+rand(2,3)*(0+1j))
 b =
 matrix columns 1 thru 3
              16.1 + 29i            1.06 + 56.1i            23.3 + 26.9i
            97.5 + 20.5i            41.4 + 79.1i            55.5 + 33.4i

> int(b)
 matrix columns 1 thru 3
                16 + 29i                 1 + 56i                23 + 26i
                97 + 20i                41 + 79i                55 + 33i
```

See also: Page 62 `ceil()`, Page 100 `floor()`, Page 189 `round()`.

int2str — integer to string conversion

int2str



`int2str` takes a scalar argument, and converts it to a string. It is intended that the argument be integer - if it has any fractional part, this will be rounded off.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `int2str.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `int2str.r` may make this function unavailable.

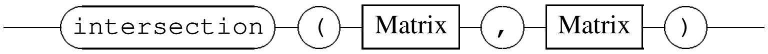
Example

```
> a = int2str(74)
74
> show(a)
   name:      a
   class:     string
    type:     string
     nr:      1
     nc:      1
> b = "74"
74
> a == b
        1
> c = int2str(73.95)
74
> show(c)
   name:      c
   class:     string
    type:     string
     nr:      1
     nc:      1
```

See also: Page 143 `num2str()`, Page 206 `sprintf()`.

intersection — Set intersection

Intersection



`intersection` calculates the intersection of the two sets supplied as arguments. The intersection of two sets is just the items that are common to both.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `intersection.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `intersection.r` may make this function unavailable.

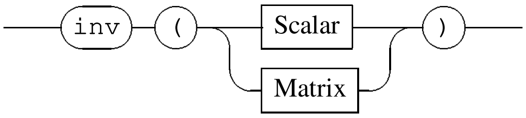
Example

```
> a = 1:6
 a =
        1          2          3          4          5          6
> b = [0,4,2,-4,7]
 b =
        0          4          2         -4          7
> intersection(a,b)
        2          4
```

See also: Page 70 `complement()`, Page 194 `set()`, Page 228 `union()`.

inv — Matrix Inverse

Inverse



`inv` computes the inverse of the argument matrix. The argument must be square and non-singular. If you specify a scalar argument, this is treated as a $1x1$ matrix, and the reciprocal of the argument is returned.

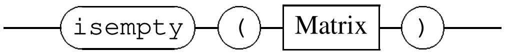
Example

```
> inv(3)
    0.333
> a = rand(3,3)
 a =
    0.908      0.879       0.98
    0.362    0.00543       0.83
    0.148      0.222    0.00526
> b = inv(a)
 b =
    -10.9       12.6       42.8
     7.14      -8.28      -23.6
      4.7      -4.23      -18.5
> a*b
        1   -1.6e-16   2.28e-15
    5e-16          1   3.16e-16
-1.32e-16   1.06e-16          1
> c = rand(3,3)+rand(3,3)*1j
 c =
 matrix columns 1 thru 3
          0.246 + 0.233i          0.461 + 0.205i          0.975 + 0.269i
          0.782 + 0.555i           0.52 + 0.561i         0.0106 + 0.334i
           0.341 + 0.29i          0.161 + 0.791i          0.414 + 0.278i
> d = inv(c)
 d =
 matrix columns 1 thru 3
         -0.314 - 0.822i           0.828 - 0.91i            0.31 + 1.28i
          0.196 + 0.801i          0.212 + 0.549i          -0.675 - 1.74i
            0.88 - 0.38i         -0.452 - 0.148i          0.313 + 0.483i
```

See also: Page 77 `det()`, Page 181 `rcond()`, Page 203 `solve()`.

isempty — Test for zero length matrix

Isempty



`isempty` returns 1 if the argument matrix is not empty. Otherwise it returns 0.

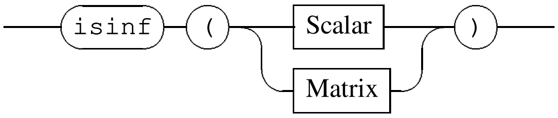
⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `isempty.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `isempty.r` may make this function unavailable.

Example

```
> isempty([2,4])
        0
> isempty([])
        1
> isempty(zeros(1,1))
        0
> isempty(zeros(0,0))
        1
> isempty(eye(0,0))
        1
> a = []
 a =
        []
> isempty(a)
        1
```

isinf — Test for Infinity

IsInf



`isinf` tests if the argument is equal to $\pm\infty$. If the test is true, it returns 1, otherwise it returns 0. If the argument is a matrix, the test is performed element-by-element, returning a matrix the same size as the original.

$\Longrightarrow$ This is not an RLAB built-in function. This function is normally loaded on start-up from the `isinf.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `isinf.r` may make this function unavailable.

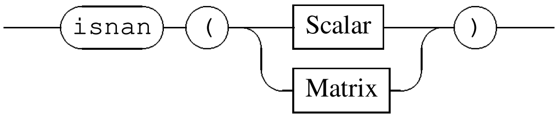
Example

```
> isinf(-inf())
       1
> isinf(inf())
       1
> isinf(nan())
       0
> isinf(0)
       0
> isinf(1)
       0
> a = [inf(),23,65,-inf(),inf()]
 a =
 Infinity         23         65  -Infinity   Infinity
> isinf(a)
       1          0          0          1          1
```

See also: Page 98 `finite()`, Page 114 `inf()`, Page 122 `isnan()`, Page 141 `nan()`,

isnan — Test for Not-A-Number

NaN



`isnan` tests if the argument is equal to NaN. If the test is true, it returns 1, otherwise it returns 0. If the argument is a matrix, the test is performed element-by-element, returning a matrix the same size as the original.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `isnan.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `isnan.r` may make this function unavailable.

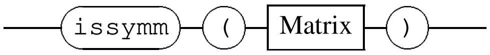
Example

```
> isnan(nan())
        1
> isnan(-nan())
        1
> isnan(inf())
        0
> isnan(0)
        0
> isnan(1)
        0
> a = [nan(),3,2,nan(),-nan()+2,45]
 a =
 matrix columns 1 thru 6
      NaN          3          2        NaN       -NaN           45

> isnan(a)
 matrix columns 1 thru 6
        1          0          0          1          1          0
```

See also: Page 98 `finite()`, Page 121 `isinf()`, Page 141 `nan()`.

issymm — Test for symmetric matrix

IsSymm



`issymm` returns 1 (true) if the argument matrix is symmetric. Otherwise it returns 0 (false).

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `issymm.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `issymm.r` may make this function unavailable.

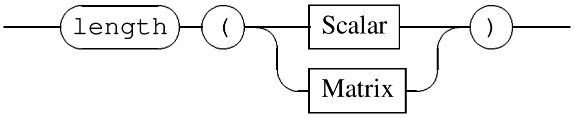
Example

```
> issymm([0,1;1,0])
        1
> issymm([0,1;1,1])
        1
> issymm([-133,1;1,1])
        1
> issymm([-133,1;1,0+346j])
        1
> issymm([-133,1;0,0+346j])
        0
```

See also: Page 217 `symm`.

length — Length of Entity

Length



The `length` function returns the length of the argument. The meaning of length varies according to the argument type:

scalar the length is 1.

string the number of characters in the string, not including the trailing null.

matrix the greater of the number of rows and the number of columns.

list the number of elements in the list.

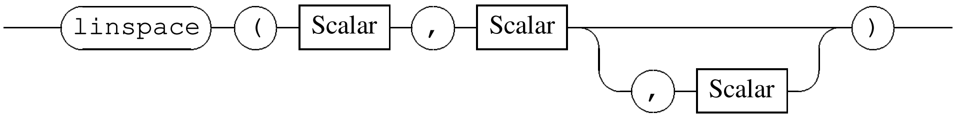
Example

```
> length(3.2567)
        1
> length("a string with (count-em) characters")
       35
> length("a string with 37(count-em) characters")
       37
> c = eye(2,5)
 c =
        1          0          0          0          0
        0          1          0          0          0
> length(c)
        5
> d = << a = 4 ; v = <<s = 3.4+2j ;c = eye(2,2)>>; z = zeros(2,2)>>
  a            v            z
> length(d)
        3
> length(d.v)
        2
```

See also: Section 195 `show()`, Section 201 `size()`.

linspace — linearly spaced vector

Linspace



`linspace` creates a vector of linearly spaced points between the first and second arguments. If the optional third argument is present, it specifies how many points are to be created. If the third argument is missing, then 100 points are assumed.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `linspace.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `linspace.r` may make this function unavailable.

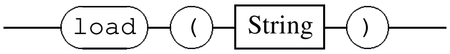
Example

```
> a = linspace(2,20);
> show(a)
   name:      a
   class:     num
   type:      real
     nr:      1
     nc:      100
> b = linspace(3.5,9.6,10)
 b =
 matrix columns 1 thru 6
      3.5       4.18       4.86       5.53       6.21       6.89

 matrix columns 7 thru 10
     7.57       8.24       8.92        9.6
> A = linspace(5,-5,13)
 A =
 matrix columns 1 thru 6
        5       4.17       3.33        2.5       1.67      0.833

 matrix columns 7 thru 12
        0     -0.833      -1.67       -2.5      -3.33      -4.17

 matrix columns 13 thru 13
       -5
> c = linspace(1,9,10)+(linspace(-4,4,10)*(0+1j))
 c =
 matrix columns 1 thru 3
                  1 - 4i            1.89 - 3.11i            2.78 - 2.22i

 matrix columns 4 thru 6
            3.67 - 1.33i           4.56 - 0.444i           5.44 + 0.444i

 matrix columns 7 thru 9
            6.33 + 1.33i            7.22 + 2.22i            8.11 + 3.11i

 matrix columns 10 thru 10
                  9 + 4i
```

See also: Page 129 `logspace()`.

load — File load

FileLoad



`load` is a function which opens the file specified by the string argument, and reads in the file's contents as if they were being typed by a user at the command line. If you prefer, this can be considered the same as temporarily redirecting the input to the file specified. The file is closed after it has been read from.

Since `load` doesn't use the normal search paths, you have to specify enough path to find the file - generally the file to be `load`ed is in the current directory, and only the name is required. However, if the file is elsewhere, you need a complete path. Often, a `rfile` command is more convenient and robust.

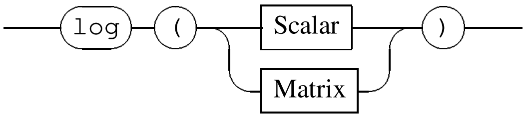
Example

```
> // Lets have a look at this file.
> system("more random.cmds");
a = rand()
b = rand(2,4);
c = "a string"
b
> load("random.cmds");
 a =
        1
 c =
a string
 b =
    0.975      0.333      0.162     0.0847
    0.647     0.0369      0.665      0.204
```

See also: Page 182 `read`.

log — natural logarithm

Log



`log` returns the natural logarithm of its argument. If the argument is a matrix an element-by-element operation is performed. If the argument is complex, then the operation returns a complex value with the real part set to the logarithm of the arguments magnitude, and the imaginary part set to the phase of the argument.

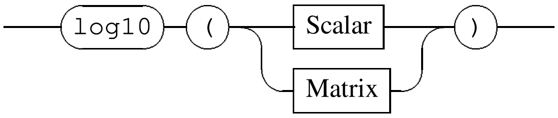
Example

```
> log(1)
        0
> b = exp(1)
 b =
     2.72
> log(b)
        1
> b^pi
     23.1
> log(b^pi)
     3.14
> a = rand(2,4)*20
 a =
       19         18       18.3       1.47
     1.84       19.2       8.82       18.5
> log(a)
     2.94       2.89       2.91      0.386
    0.608       2.95       2.18       2.92
```

See also: Page 128 `log10()`.

log10 — Base 10 logarithm

Log10



`log10` returns the base-10 logarithm of it's argument. If the argument is a matrix, an element-by-element `log10` operation is performed. If the argument is complex, this is the same as taking natural logarithm of the argument, then multiplying by the natural logarithm of 10.

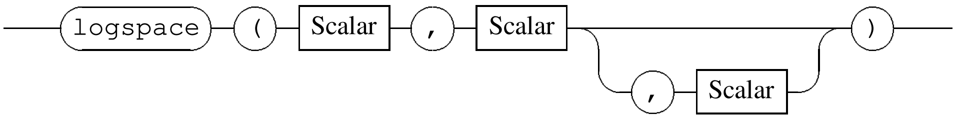
Example

```
> log10(1)
        0
> log10(10)
        1
> log10(100)
        2
> log10(100^4)
        8
> log10(10^3.52464)
     3.52
> a = rand(2,5)*25
 a =
     6.16       8.52         13       24.4       10.4
     19.6       11.5       4.02      0.265       5.82
> log10(a)
    0.789       0.93       1.11       1.39       1.02
     1.29       1.06      0.604     -0.576      0.765
```

See also: Page 127 `log()`.

logspace — Logarithmically spaced vector

Logspace



`logspace` creates a vector of logarithmically spaced points. If the first two arguments are x_1 and x_2, then the points are generated between 10^{x_1} and 10^{x_2}. If the optional third argument is present, it specifies how many points are to be created. If the third argument is missing, then 50 points are assumed.

As a special case, if the second argument is equal to π, then the points are generated between 10^{x_1} and π.

$\Longrightarrow$ This is not an RLAB built-in function. This function is normally loaded on start-up from the `logspace.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `logspace.r` may make this function unavailable.

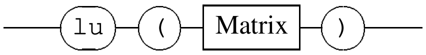
Example

```
> logspace(0,1)
 matrix columns 1 thru 7
        1       1.05        1.1       1.15       1.21       1.26       1.33

 matrix columns 8 thru 14
     1.39       1.46       1.53        1.6       1.68       1.76       1.84

 matrix columns 15 thru 21
     1.93       2.02       2.12       2.22       2.33       2.44       2.56

 matrix columns 22 thru 28
     2.68       2.81       2.95       3.09       3.24       3.39       3.56

 matrix columns 29 thru 35
     3.73       3.91       4.09       4.29        4.5       4.71       4.94

 matrix columns 36 thru 42
     5.18       5.43       5.69       5.96       6.25       6.55       6.87

 matrix columns 43 thru 49
      7.2       7.54       7.91       8.29       8.69        9.1       9.54

 matrix columns 50 thru 50
       10
> logspace(1,3,10)
 matrix columns 1 thru 7
       10       16.7       27.8       46.4       77.4        129        215

 matrix columns 8 thru 10
      359        599     1e+03
> logspace(0,pi,10)
 matrix columns 1 thru 7
        1       1.14       1.29       1.46       1.66       1.89       2.15

 matrix columns 8 thru 10
     2.44       2.77       3.14
```

See also: Page 125 `linspace()`.

lu — LU decomposition

LU



`lu` performs an LU decomposition of the matrix argument. The input matrix must be square and non-singular. `lu` returns the lower, upper and pivot matrices as the `l`, `u` and `pvt` elements of a list. If the argument matrix is A, then the decomposition has the form:

$$A = pvtlu$$

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `lu.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `lu.r` may make this function unavailable.

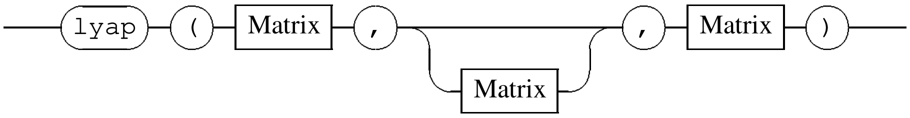
Example

```
> a = rand(4,4)
 a =
    0.957      0.367      0.641      0.252
     0.89      0.638      0.245      0.955
    0.279       0.18      0.284      0.921
     0.42      0.589      0.869      0.376
> b = lu(a)
   l            pvt          u
> b.l
 l =
        1          0          0          0
    0.439          1          0          0
     0.93      0.693          1          0
    0.291      0.171    0.00416          1
> b.u
 u =
    0.957      0.367      0.641      0.252
        0      0.428      0.589      0.265
        0          0     -0.758      0.537
        0          0          0        0.8
> b.pvt
 pvt =
        1          0          0          0
        0          0          1          0
        0          0          0          1
        0          1          0          0
> b.pvt*b.l*b.u
    0.957      0.367      0.641      0.252
     0.89      0.638      0.245      0.955
    0.279       0.18      0.284      0.921
     0.42      0.589      0.869      0.376
```

See also: Page 58 `backsub()`, Page 92 `factor()`, Page 119 `inv()`, Page 203 `solve()`.

lyap — solution of the lyapunov equation

Lyapunov



`lyap` is used to solve the general form of the Sylvester equation, and its special form in the Lyapunov equation. It is very similar to the `sylv` function, except that the arguments need not be upper triangular.

Given the first matrix argument is A, the second matrix argument is B, and the third matrix argument is C, `lyap` solves

$$Ax + xB = -C$$

for x.

If there are only two arguments, it is assumed $B = A^T$, and the Lyapunov equation is solved. Note that in `sylv`, you leave out the comma and the argument — `lyap` requires both commas.

$\Longrightarrow$ This is not an RLAB built-in function. This function is normally loaded on start-up from the `lyap.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `lyap.r` may make this function unavailable.

Example

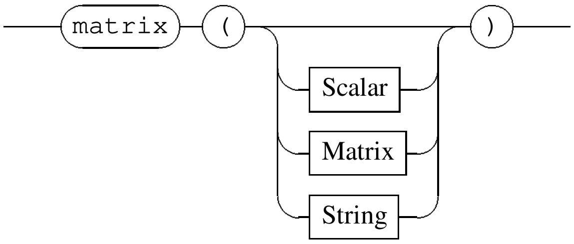
```
> a = rand(3,3)
 a =
    0.341      0.455      0.188
    0.623     0.0178      0.946
    0.182      0.071      0.799
> b = rand(3,3)
 b =
    0.369      0.481      0.931
    0.895      0.791       0.98
    0.836      0.733      0.195
> c = rand(3,3)
 c =
    0.459       0.46      0.749
   0.0589      0.225      0.385
    0.281      0.957      0.642
> ResSylv = lyap(a,b,c)
 ResSylv =
   -0.804      0.868      -1.24
    0.583     0.0114     -0.375
     0.55     -0.825    -0.0942
> a*ResSylv+ResSylv*b + c
 4.44e-16   8.88e-16   1.44e-15
 5.55e-17  -1.11e-16   6.66e-16
 2.22e-16          0   2.22e-16
> ResLyap = lyap(a,,c)
 ResLyap =
    -3.75       7.11     -0.916
    -3.07       2.15       1.22
     2.29       -4.2     -0.427
> a*ResLyap+ResLyap*a' + c
```

```
        0   8.88e-16   2.55e-15
 1.94e-15  -2.39e-15   1.11e-16
-1.11e-15  -1.11e-16  -1.33e-15
```

See also: Page 193 `schur()`, Page 216 `sylv()`.

matrix — Convert to Matrix

Matrix



The `matrix` function attempts to convert its argument to a matrix. If you pass no arguments, you get an empty matrix back. If you pass a scalar or string argument, you get back a one element matrix with the argument as that element. If you pass a matrix argument, you just get that matrix back.

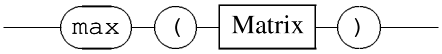
Example

```
> matrix()
        []
> matrix(3)
        3
> show(matrix(3))
   name:      NULL
   class:     num
   type:      real
     nr:      1
     nc:      1
> show(matrix("This is a test string"))
   name:      NULL
   class:     string
   type:      string
     nr:      1
     nc:      1
> a = [2.4,5,3,6;2,-6,-9,102]
 a =
      2.4          5          3          6
        2         -6         -9        102
> matrix(a)
      2.4          5          3          6
        2         -6         -9        102
```

See also: Page 191 `scalar()`.

max — Maximum Value

Max



`max` returns the maximum value contained in the argument matrix. If the argument is a row or column vector, then the largest value is returned. If the argument is a not a vector, then a row vector is returned, containing the maximum value from each column of the argument. If the argument is complex, magnitudes are used.

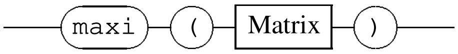
Example

```
> c = rand(3,3)
 c =
    0.214       0.87      0.352
    0.316       0.11      0.226
    0.402      0.617      0.217
> max(c)
    0.402       0.87      0.352
> d = rand(1,3)
 d =
    0.622      0.356      0.156
> max(d)
    0.622
> e = rand(3,1)
 e =
    0.802
    0.588
    0.022
> max(e)
    0.802
```

See also: Page 135 `maxi()`, Page 138 `min()`, Page 139 `mini()`.

maxi — Index of maximum value

MaxI



`maxi` returns the index of the maximum value contained in matrix. If the argument is a row or column vector, then the index of the largest value is returned. If the argument is not a vector, then a row vector of the column indices of the largest column values of is returned. If the argument is complex, magnitudes are compared.

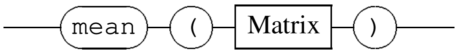
Example

```
> a = rand(3,4)
 a =
        1      0.333      0.665      0.167
    0.975     0.0369     0.0847      0.655
    0.647      0.162      0.204      0.129
> maxi(a)
        1          1          1          2
> a = rand(3,5)
 a =
     0.91      0.265     0.0918      0.915      0.924
    0.112        0.7      0.902      0.441     0.0882
    0.299       0.95       0.96     0.0735      0.908
> maxi(a)
        1          3          3          1          1
> maxi(a’)
        5          3          3
```

See also: Page 134 `max()`, Page 138 `min()`, Page 139 `mini()`.

mean — Average value

Mean



`mean` calculates the arithmetic mean of the argument. If the argument is a row or column vector, then the mean of all elements is returned, Otherwise `mean` returns a row matrix formed by taking the arithmetic mean of each column.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `mean.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `mean.r` may make this function unavailable.

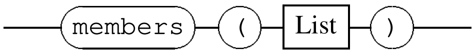
Example

```
> a = rand(3,5)
 a =
    0.572      0.109      0.929      0.281      0.436
    0.576      0.902      0.935      0.972      0.625
    0.501      0.568      0.771     0.0269      0.469
> mean(a)
     0.55      0.526      0.878      0.427       0.51
> b = a[1;]
 b =
    0.572      0.109      0.929      0.281      0.436
> mean(b)
    0.465
> c = a[;1]
 c =
    0.572
    0.576
    0.501
> mean(c)
     0.55
```

See also: Page 210 `std()`.

members — Items in a List

Members



`members` is a function that takes a list argument, and returns a string matrix containing the names of all the elements in that list. This string matrix can then be used to reference all the elements in the list.

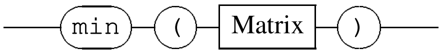
Example

```
> mylist = << str1 = "a string"; mat = eye(3,3); str2 = "str2.5" >>
 mylist =
   mat             str1            str2
> mylist.scalar1 = 2.2354e-23
 mylist =
   mat             scalar1         str1            str2
> members(mylist)
mat      scalar1  str1     str2
> for (elem in members(mylist))
  {
  printf("name:%s\n",elem);
  mylist.[elem]
  }
name:mat
 mat =
        1          0          0
        0          1          0
        0          0          1
name:scalar1
 scalar1 =
 2.24e-23
name:str1
 str1 =
a string
name:str2
 str2 =
str2.5
```

See also: Page 229 `what()`, Page 231 `who()`.

min — Minimum Element

Min



`min` returns the minimum value or values contained in the matrix argument. If the argument is a row or column vector, then the smallest value is returned. If the argument is not such a vector, then a row-vector is returned containing the minimum value from each column of the argument. The minimum value of a complex argument is found by comparing magnitudes.

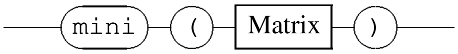
Example

```
> a = rand(3,5)
 a =
    0.362    0.00543       0.83      0.782       0.52
    0.148      0.222    0.00526      0.341      0.161
    0.879       0.98      0.246      0.461      0.975
> min(a)
    0.148    0.00543    0.00526      0.341      0.161
> min(a')'
  0.00543
  0.00526
    0.246
```

See also: Page 134 `max()`, Page 135 `maxi()`, Page 139 `mini()`.

mini — Index of Minimum Value

MiniI



`mini` returns the index of the minimum value contained in the argument matrix. If the argument is a row or column vector, then the index of the smallest value is returned. If the argument is not such a vector, then a row vector of the column indices of the smallest value in each column is returned.

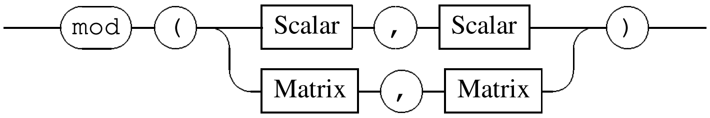
Example

```
> a = rand(3,5)
 a =
   0.0106      0.555      0.561      0.334      0.692
    0.414       0.29      0.791      0.278      0.737
    0.233      0.205      0.269      0.789      0.248
> mini(a)
        1          3          3          2          3
> mini(a')'
        1
        4
        2
```

See also: Page 134 `max()`, Page 135 `maxi()`, Page 138 `min()`.

mod — Remainder after division

Mod



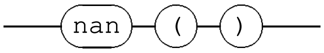
`mod` returns the floating point remainder of the division of the first argument by the second argument. If the divisor is zero or if the division would overflow, then it returns zero. When the arguments `mod` are both matrices, then an element by element operation is performed, so they must have the same dimensions.

Example

```
> mod(7,2)
        1
> mod(8,3)
        2
> mod(17,6)
        5
> mod(18,6)
        0
> Mat1 = round(rand(2,5)*100)
 Mat1 =
       65         91         30         70          9
       13         11         27         95         90
> Mat2 = round(rand(2,5)*10)
 Mat2 =
        3          3          7          2          4
        3          8          7          5          4
> mod(Mat1,Mat2)
        2          1          2          0          1
        1          3          6          0          2
```

nan — Not-a-Number Value

NotANumber



`nan` returns a scalar encoded with the IEEE-754 NaN (Not a Number) value. You should be able to use this value as an argument to just about any operation or function, with a result of NaN.

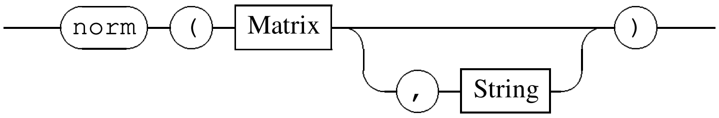
Example

```
> nan()
      NaN
> nan() - nan()
      NaN
> nan() + nan()
      NaN
> exp(nan())
      NaN
```

See also: Page 114 `inf()`, Page 122 `isnan()`.

norm — Matrix Norm

Matrix



`norm` can calculate various types of matrix norms. The type of norm to be calculated is specified by the optional string matrix, with a defult of 1-norm.

`"M"` or `"m"` returns the absolute value of the element with the largest magnitude in the matrix.

`"1"`, `"O"` or `"o"` returns the 1-norm. This is defined as the maximum absolute column sum.

`"2"` returns the matrix 2-norm. This is the largest singular value in the matrix argument.

`"I"` or `"i"` returns the infinity-norm. This is defined as the maximum absolute row sum.

`"F"`, `"f"`, `"E"` or `"e"` returns the Frobenius norm. This is the square root of the of the sum of squares of all the elements in the matrix argument.

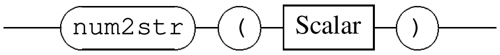
Example

```
> a = rand(5,5)
 a =
    0.269      0.737      0.505     0.0321      0.324
    0.334      0.248      0.529      0.411      0.983
    0.278       0.45      0.877      0.982      0.734
    0.789      0.363      0.465      0.686      0.539
    0.692      0.369      0.806      0.447      0.163
> sum(a)
     2.36       2.17       3.18       2.56       2.74
> norm(a)
     3.18
> norm(a,"o")
     3.18
> sum(a’)’
     1.87
     2.51
     3.32
     2.84
     2.48
> norm(a,"i")
     3.32
> norm(a,"m")
    0.983
> svd(a).sigma
 sigma =
     2.68      0.723      0.606      0.465      0.256
> norm(a,"2")
     2.68
> sqrt(sum(sum(a.*a)))
     2.89
> norm(a,"f")
     2.89
```

See also: Page 134 `max()`. Page 213 `sum()`, Page 214 `svd()`.

num2str — Number to string conversion

NumToStr



`num2str` converts a real scalar into a $1x1$ string matrix. This may result in scientific notation if the scalar is particularly large or small. An error will result if the argument is not a real scalar.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `num2str.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `num2str.r` may make this function unavailable.

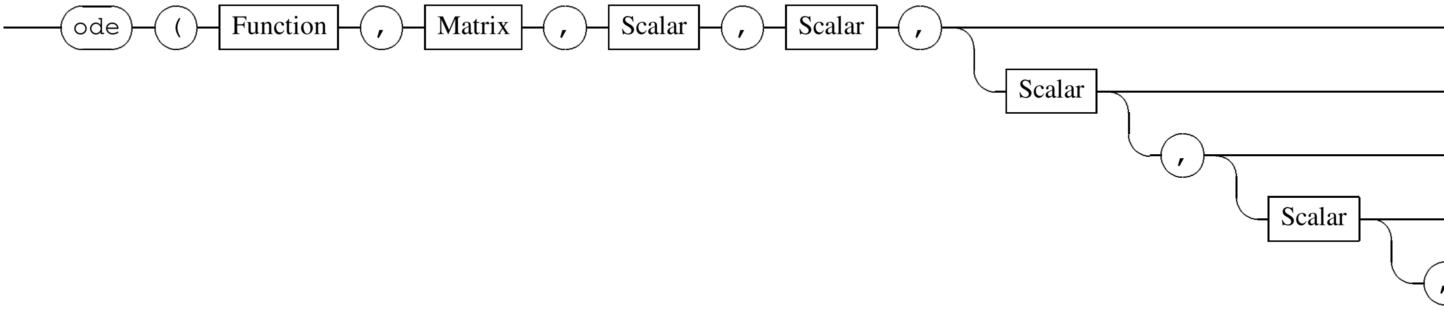
Example

```
> a = 3.4321
 a =
     3.43
> b = num2str(a)
3.432
> show(b)
   name:      b
   class:     string
    type:     string
     nr:      1
     nc:      1
> num2str(298649678)
2.986e+08
> show(num2str(298649678))
   name:      NULL
   class:     string
    type:     string
     nr:      1
     nc:      1
```

See also: Page 117 `int2str()`, Page 206 `sprintf()`.

ode — ordinary differential equation solver

ODE



`ode` is a first order Ordinary Differential Equation solver. It integrates *sets* of first order differential equations.

```
                dy(i)/dt = f(t,y(1),y(2),...,y(N))
                y(i) given at  t .

Syntax: ode ( rhsf, ystart, tstart, tend, dtout, relerr, abserr )

        rhsf    A function that evaluates dy(i)/dt at t. The function
                takes two arguments and returns dy/dt. An example that
                generates dy/dt for Van der Pol's equation is shown
                below.
Can be user or bltin function.

        ystart  The initial values of y, y(tstart).
Must be column or row vector
        tstart  The initial value of the independent variable.
Can also be a matrix - first element will be used.
        tend    The final value of the independent variable.
Can also be a matrix - first element will be used.
Cannot be same as tstart

        dtout   The output interval. The vector y will be saved at
                tstart, increments of tstart + dtout, and tend. If
                dtout is not specified, then the default is to store
                output at 101 values of the independent variable.

        relerr  The relative error tolerance. Default value is 1.e-6.

        abserr  The absolute error tolerance. At each step, ode
                requires that:

                abs(local error) <= abs(y)*relerr + abserr

                For each component of the local error and solution
                vectors. The default value is 1.e-6.
```

```
The Fortran source code for ode() is completely explained and
documented in the text, "Computer Solution of Ordinary
Differential Equations: The Initial Value Problem" by
L. F. Shampine and  M. K. Gordon.

Example:

vdpol = function ( t , x )
{
  local (xp)
  xp = zeros(2,1);
  xp[1] = x[1] * (1 - x[2]^2) - x[2];
  xp[2] = x[1];
  return xp;
};

x0 = [0; 0.25];

xbase = ode( vdpol, x0, 0, 20, 0.05, 1e-9, 1e-9);
```

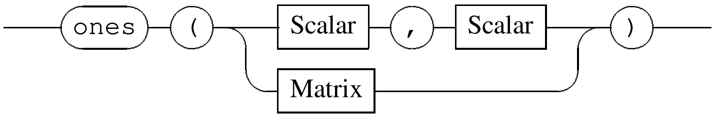
Example

`Dummy Example`

See also: Page 1 `()`.

ones — Matrix of ones

Ones



`ones` creates a matrix with each element set to 1. The arguments specify the dimensions of the matrix to be created. If the argument is a matrix, the matrix must have two elements, with the first argument specifying the number of rows, and the second argument specifying the number of columns. This is the same format as the output of the `size` function. If the arguments are two scalars, then the first argument is the number of rows, and the second argument is the number of columns.

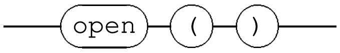
Example

```
> ones(2,5)
        1          1          1          1          1
        1          1          1          1          1
> ones([2,4])
        1          1          1          1
        1          1          1          1
> d = 1:3:0.5;
> c = 1:5;
> b = [c;d]
 b =
        1          2          3          4          5
        1        1.5          2        2.5          3
> a = ones(size(b))
 a =
        1          1          1          1          1
        1          1          1          1          1
> ones([2,4])
```

See also: Page 201 `size()`, Page 239 `zeros()`.

open —

Open



`open`

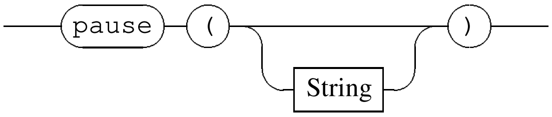
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

pause — Pause program

Pause



`pause` displays `Hit return to continue`, and then stops execution of an RLAB program until the return key is pressed. If the optional string argument is present, it is used instead of the `Hit return to continue` message.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `pause.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `pause.r` may make this function unavailable.

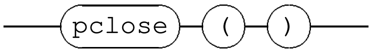
Example

```
> pause()
        Hit return to continue

        0
> pause();
        Hit return to continue

> pause("Kindly press any key:");
Kindly press any key:
j
> pause("Kindly press the return key:");
Kindly press the return key:

> pause("Hit return, or else!");
Hit return, or else!

>
```

See also: Page 102 `fprintf()`, Page 108 `getline()`.

pclose — close plot window

PClose



`pclose` closes the current plot window.

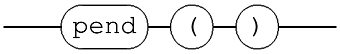
Example

```
No example yet - coming soon
```

See also: Page 150 `pend()`,

pend — Close all plotting windows

PEnd



`pend` is used to end the plotting session. This is useful when you have many plot windows open and do not wish to use `pclose` on each one induvidually.

Note: You can begin plotting again by using `pstart`.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `plot.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `plot.r` may make this function unavailable.

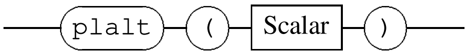
Example

```
> pend()
        0
> pend();
```

See also: Page 68 `pclose()`.

plalt — Set viewing altitude

PlAlt



`plalt` is used to set the viewing altitude for three dimensional plots. This is expressed in degrees above the XY plane.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `plot.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `plot.r` may make this function unavailable.

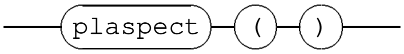
Example

```
No example yet - coming soon
```

See also: Page 154 `plaz()`.

plaspect —

PlAspect



```
plaspect
```

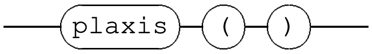
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plaxis —

PlAxis



```
plaxis
```

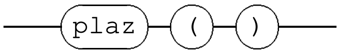
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plaz —

PlAz



`plaz`

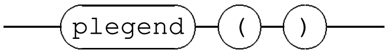
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plegend —

Plegend



`plegend`

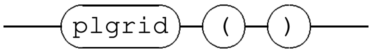
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plgrid —

PlGrid



```
plgrid
```

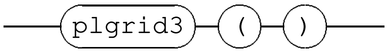
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plgrid3 —

PlGrid3



```
plgrid3
```

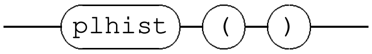
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plhist —

PlHist



```
plhist
```

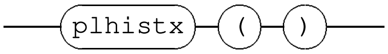
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plhistx —

Plhistx



```
plhistx
```

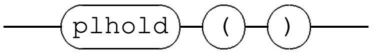
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plhold —

PlHold



`plhold`

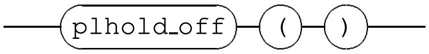
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plhold_off —

PlHoldOff



```
plhold_off
```

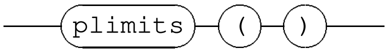
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plimits —

PLimits



```
plimits
```

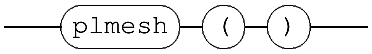
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plmesh —

Plmesh



`plmesh`

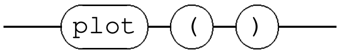
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plot —

Plot



`plot`

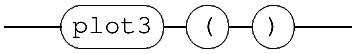
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plot3 —

Plot3



```
plot3
```

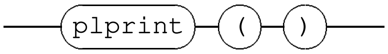
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plprint —

PlPrint



```
plprint
```

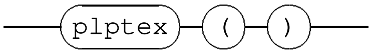
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plptex —

PlpTeX



```
plptex
```

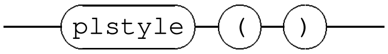
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plstyle —

PlStyle



`plstyle`

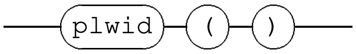
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

plwid —

Plwid



```
plwid
```

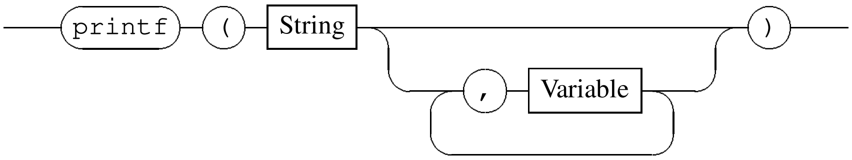
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

printf — Formatted Output

FormatPrint



The `printf` function is intended for writing formatted output to standard output, which is usually the screen. It is similar to the C language function of the same name, though some features are limited or not available, since RLAB doesn't have all the data types of C.

The first argument is the format string. It consists of the text to be written out, and possibly some conversion specifications. A conversion specification is a sequence of commands that determine how the remainder of the arguments are to be displayed. The left most conversion specifier is matched to the third argument, the next specifier to the fourth argument, and so on. Each argument has to be matched to a conversion specification. When the format string is being scanned, the argument that matches the the specifier that the scanner is up to is called the *current argument* .

Conversion specifications always begin with a `%` sign. The next thing that can occur are the flags:

- causing the conversion to be left-justified, instead of the default right-justified.

+ causing a sign to always be prepended to the conversion, instead of the default of only prepending minus signs.

which causes the format to be of an alternate form. This only has meaning for `e`, `E`, `f`, `g` and `G` formats, where is causes a decimal point to be used always. It also prevents suppression of trailing zeros for `g` and `G` options.

0 (zero) which causes padding by leading zeros, instead of the default space padding. This is overridden by both − and specifying a precision.

space which causes a space to be prepended if no sign is present. This is overridden by the + flag.

Following any flags, a minimum field width may be specified. This is either a integer constant, or a `*` character. If it is a constant, this is the minimum width. If it is a `*`, then the current argument (which must be a scalar) is used to specify the minimum width. If there is an optional width, then there may also be an option precision. This is specified in the same way as the width, using either and integer constant, or a `*` to signify that the current argument is to be used.

The next thing that can occur is an optional `h` or `l` (ell) modifier. This changes the behavior of the subsequent `i` and `u` conversion specifiers. It is legal, but has no effect, with the `d` specifier. It is illegal with all other specifiers.

The final thing that must occur is a character specifying the way in which the current variable is to be displayed. The valid characters are

c causes the argument to be converted to an unsigned character format.

d which causes the output to be displayed as a decimal string of the form *[-]dddd*. The precision specifies the minimum number of digits to appear, with a default of one.

e causes the output to be displayed in scientific notation of the form *[-]d.ddd*e±*dd*, where *d* is any digit. There is always one non-zero digit before the decimal place if the argument is non-zero. The precision specification sets how many digits are present after the decimal place, with a default of six.

E is the same as `e`, except that a upper case letter `E` is used to seperate the mantissa and exponent instead of a lower case `e`.

f causes the output to be displayed as a decimal string of the form *ddd.ddd*. the precision specifies how many digits should appear after the decimal place, with a default of six.

g is the same as `e` is the exponent would be less than -4, or greater than the precision. Otherwise, it is the same as `f`.

G is the same as `E` is the exponent would be less than -4, or greater than the precision. Otherwise, it is the same as `f`.

i is the same as `d`, except that the `l` flag may be used.

s may be used to display strings. If the argument is a scalar, then this is the same as `f`.

u displays the argument as an unsigned integer value. The precision specifies the minimum number of digits to be displayed, with a default value of one.

The number of arguments must match the number of conversion specifications, including those required for width and precision specifications.

`printf` cannot print out whole matrices or lists. `write` knows how to deal with entire data objects.

Example

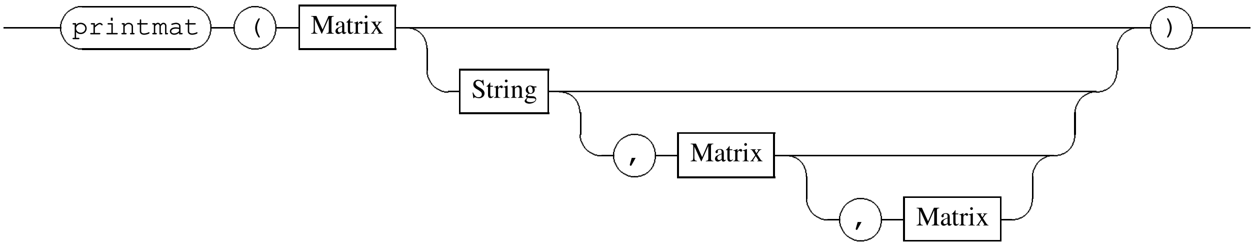
```
> a = rand()
 a =
     0.95
> b = rand() + rand()*1j
 b =
    0.244 + 0.162i
> c = "der string"
 c =
der string
> printf("%e , %g , %d", a,a,a)
9.501680e-01 , 0.950168 , 0       27
> printf("%e , %g , %d", a,a,a);
9.501680e-01 , 0.950168 , 0> printf("%e , %g , %d\n", a,a,a);
9.501680e-01 , 0.950168 , 0
> printf("%+15e , %-f , %s\n", a,b,c);
  +9.501680e-01 , 0.244271 , der string
> d = rand(3,4)
 d =
    0.252      0.985     0.0747      0.462
    0.337      0.811      0.482      0.684
    0.302      0.702     0.0618      0.351
> for(i in 1:size(d)[1]) {
    for(j in 1:size(d)[2]) {
      printf("d[%i;%i] = %f\n", i , j, d[i;j]);
    }
  }
d[1;1] = 0.251599
d[1;2] = 0.984848
d[1;3] = 0.074694
d[1;4] = 0.461953
d[2;1] = 0.336841
d[2;2] = 0.811462
d[2;3] = 0.482371
```

```
d[2;4] = 0.684488
d[3;1] = 0.301645
d[3;2] = 0.702280
d[3;3] = 0.061787
d[3;4] = 0.351217
```

See also: Page 102 `fprintf()`, Page 182 `read()`, Page 206 `sprintf()`, Page 233 `write()`.

printmat — Pretty print a matrix

PrintMat



`printmat` prints the matrix specified by the first matrix argument. If the optional string argument is supplied, it is used as the title — othewise no label is used. The next argument is a string vector containing labels for each row, and the final argument is a string vector containing labels for each column. If the row or column labels are missing, labels are generated.

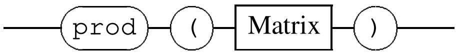
⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `printmat.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `printmat.r` may make this function unavailable.

Example

```
> a = rand(4,3)
 a =
        1     0.0369      0.204
    0.975      0.162      0.167
    0.647      0.665      0.655
    0.333     0.0847      0.129
> ColLabels = ["Col 1","2nd","The third Column"];
> RowLabels = ["Row 1","second row","beavis","butthead"];
> printmat(a)
              ----1----   ----2----   ----3----
--  1 -->        1.00000     0.03694     0.20414
--  2 -->        0.97452     0.16171     0.16731
--  3 -->        0.64748     0.66465     0.65496
--  4 -->        0.33309     0.08467     0.12882

        0
> printmat(a, "My very own label", RowLabels, ColLabels);

My very own label =

                   Col 1         2ndThe third Column
Row 1            1.00000     0.03694     0.20414
second row       0.97452     0.16171     0.16731
beavis           0.64748     0.66465     0.65496
butthead         0.33309     0.08467     0.12882
```

prod — Product of matrix elements

Prod



If the argument is a row or column vector, `prod` computes the product of the elements of that vector. If the argument is not a vector, a row vector containing the product of each column is returned.

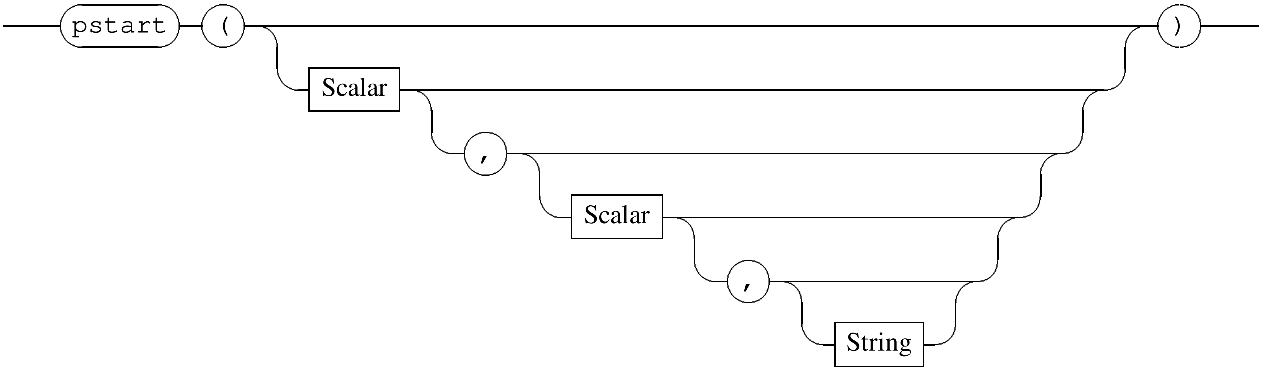
Example

```
> d = 1:3:0.5
 d =
        1        1.5          2        2.5          3
> prod(d)
     22.5
> e = d'
 e =
        1
      1.5
        2
      2.5
        3
> prod(e)
     22.5
> f = 10*rand(2,5)
 f =
       10       6.47      0.369       6.65       2.04
     9.75       3.33       1.62      0.847       1.67
> prod(f)
     97.5       21.6      0.597       5.63       3.42
> prod(f')
      325       74.4
> prod(prod(f))
 2.41e+04
```

See also: Page 75 `cumprod()`, Page 213 `sum()`.

pstart — Create main plot window

PStart



`pstart` is used to create a plotting window. Within each plotting window there can be sub-plots. The first argument to `pstart` is the number of sub-plots to be created in the horizontal direction, and the second argument is the number of sub-plots to be created in the vertical direction. If these arguments are omitted, they default to 1. This means that if both are omitted, a single sub-plot wil be created within the plotting window. The third argument is the type of output. The list of valid output devices is dependent on the setup of the plotting library for your particular system, however some typical values include `"xwin"` — X Window System, `"xterm"` — X terminal, `"plmeta"` — PLPLOT portable meta-file format, `"tekt"` — Tektronix terminal, `"xfig"` — Xfig file, `"ps"` — Postscript, `"psc"` — Colour Postscript and `"hp7470"` —- HPGL plotter. To obtain a full list, execute `pstart` without any arguments.

The newly created plot window will become the current plot window.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `plot.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `plot.r` may make this function unavailable.

Example

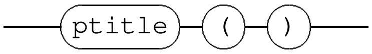
```
> pstart();

Plotting Options:
 < 1> (xwin)     X-Window (Xlib)
 < 2> (xterm)    Xterm Window
 < 3> (tekt)     Tektronix Terminal (4010)
 < 4> (t4107t)   Tektronix Terminal (4105/4107)
 < 5> (mskermit)         MS-Kermit emulator
 < 6> (dg300)    DG300 Terminal
 < 7> (plmeta)   PLPLOT Native Meta-File
 < 8> (tekf)     Tektronix File (4010)
 < 9> (t4107f)   Tektronix File (4105/4107)
 <10> (ps)       PostScript File (monochrome)
 <11> (psc)      PostScript File (color)
 <12> (xfig)     Xfig file
 <13> (ljii)     LaserJet II Bitmap File (150 dpi)
 <14> (hp7470)   HP 7470 Plotter File (HPGL Cartridge, Small Plotter)
 <15> (hp7580)   HP 7580 Plotter File (Large Plotter)
 <16> (imp)      Impress File
 <17> (null)     Null device
```

```
Enter device number or keyword: 17
Sending output to Null device..
> pstart(1,2,"hp7470")
Enter desired name for graphics output file: dummy.hpgl
Created dummy.hpgl
 P =
        1
```

See also: Page 149 `pclose()`, Page 150 `pend()`, Page 176 `pwin()`.

ptitle —

PTitle



`ptitle`

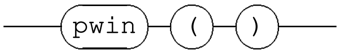
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

pwin —

PWin



`pwin`

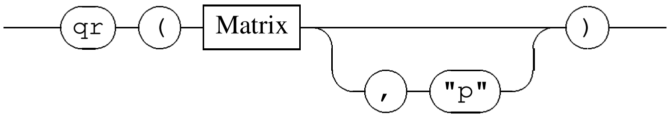
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

qr — QR decomposition

QR



`qr` computes the QR decomposition of the argument matrix, returning a list with two elements `q` and `r`. These are the Q and R matrices, such that, for an argument matrix of A:

$$A = QR$$

If the optional argument `"p"` is used, column pivoting is used, with the permutation matrix returned in the list result as `p`, such that for a permutation matrix P,

$$AP = QR$$

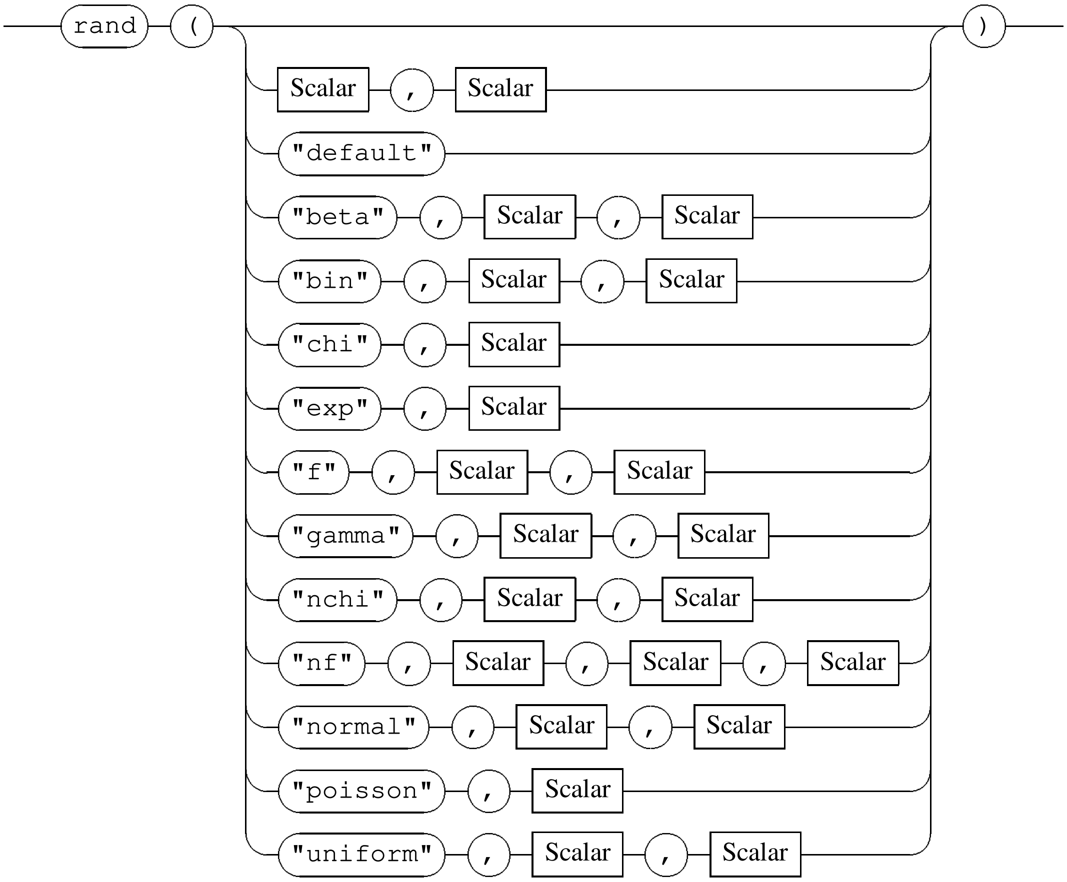
Q will have orthogonal columns, and R will be upper-triangular.

Example

```
> a = rand(5,5)
 a =
     0.81      0.621      0.328     0.0767      0.818
    0.909      0.845      0.799      0.647      0.367
    0.293      0.132     0.0398     0.0221      0.191
    0.743     0.0376      0.398      0.227      0.845
    0.705      0.811      0.263      0.431      0.401
> res = qr(a)
 res =
   q            r
> a - res.q*res.r
-1.11e-16  -3.33e-16  -3.89e-16  -1.11e-16  -2.22e-16
        0          0   1.11e-16   1.11e-16  -1.67e-16
        0  -5.55e-17  -9.71e-17  -1.42e-16  -2.78e-17
        0  -6.94e-17  -2.22e-16  -5.55e-17          0
        0          0  -5.55e-17  -5.55e-17  -5.55e-17
> res2 = qr(a,"p")
 res2 =
   p            q            r
> a*res2.p - res2.q*res.r
-1.11e-16  -3.33e-16  -3.89e-16  -1.11e-16  -2.22e-16
        0          0   1.11e-16   1.11e-16  -1.67e-16
        0  -5.55e-17  -9.71e-17  -1.42e-16  -2.78e-17
        0  -6.94e-17  -2.22e-16  -5.55e-17          0
        0          0  -5.55e-17  -5.55e-17  -5.55e-17
```

rand — Random Values

Random



`rand` has two distinct forms. The first form is used to set the distribution that is used to generate random values. The second form actually generates the values. We will look at each form in turn.

- β distribution. If the first scalar argument is a and the second scalar argument is b, then the density of the distribution is
$$\frac{x^{a-1}(1-x)^{b-1}}{B(a,b)} \text{ for } 0 < x < 1$$
where $B(a,b)$ is the binomial distribution.
- Binomial distribution. The first scalar argument specifies the number of trials, and the second scalar argument specifies the probability of an event occuring in a trial.
- χ^2 distribution. The scalar argument specifies the number of degrees of freedom.
- Default distribution. The default distribution is a uniform distribution between 0 and 1.
- Exponential distribution. The scalar argument specifies the mean of the distribution.
- $\mathcal{F}$ distribution. The first scalar argument specifies the degrees of freedom present in the numerator, and the second scalar argument specifies the degrees of freedom in the denominator.
- Γ distribution. If the first scalar argument is a, and the second scalar argument is r, then the density of the distribution is:
$$\frac{a^r}{\Gamma(r)} x^{r-1} e^{-ax}$$

- Non-central χ^2 distribution The first scalar argument specifies the number of degrees of freedom, and the second scalar argument specifies the non-centrality.
- Non-central $\mathcal{F}$ distribution. The first scalar argument specifies the degrees of freedom present in the numerator, the second scalar argument specifies the degrees of freedom in the denominator, and the third scalar argument specifies the non-centrality.
- Normal distribution. The first scalar argument specifies the mean, and the second scalar argument specifies the standard distribution.
- Poisson distribution. The scalar argument specifies the mean of the distribution.
- Uniform distribution. The first scalar argument specifies the lower limit, and the second scalar argument specifies the upper limit.

The second form of the `rand` function is used to produce random variables from whatever distribution is actually selected at the time. If you supply no arguments, then a random scalar is returned. If you supply two scalar arguments, then a matrix is returned. The first argument specifies the number of rows, and the second argument specifies the number of columns. In addition, you can supply a matrix argument, which must have two elements, where the first is the number of rows, and the second is the number of columns. This is intended for use with the `size` function.

If `srand` is not used to set the seed value, each RLAB session will produce the same results.

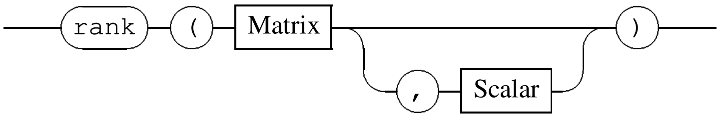
Example

```
> rand()
        1
> rand(2,4)
    0.975       0.333       0.162      0.0847
    0.647      0.0369       0.665       0.204
> rand(3,2)+rand(3,2)*1j
          0.167 + 0.265i          0.91 + 0.0918i
            0.655 + 0.7i          0.112 + 0.902i
           0.129 + 0.95i           0.299 + 0.96i
> rand("bin",100,0.5)
        1
> rand(2,5)
       44          56          57          52          56
       44          46          48          44          46
> rand("normal",0,1)
        1
> rand(2,5)
    0.495      -0.757       0.909      -0.748      -0.586
   -0.115       0.684       0.713      -0.456       0.619
```

See also: Page 209 `srand()`.

rank — Rank of a Matrix

Rank



`rank` calculates the rank of the argument matrix. The rank of a matrix is the number of non-zero singular values of that matrix. However since numeric computation introduces errors, the singular values are actually tested against a set tolerance value. The optional scalar argument, if present, specifies the tolerance, otherwise the default of the triple-product of the greater dimension of the argument, the 2-norm of the matrix and machine epsilon is used.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `rank.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `rank.r` may make this function unavailable.

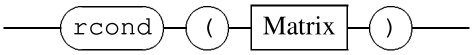
Example

```
> a = rand(5,5)
 a =
     0.44      0.741      0.842      0.599      0.239
    0.685      0.313      0.652      0.455      0.465
    0.129      0.759      0.386      0.658      0.546
     0.79      0.151      0.583      0.232      0.801
    0.868      0.325      0.205      0.797      0.247
> svd(a).sigma
 sigma =
      2.6      0.835      0.624      0.442     0.0475
> rank(a)
        5
> b = [a,a;a,a];
> svd(b).sigma
 sigma =
 matrix columns 1 thru 6
      5.2       1.67       1.25      0.884      0.095   1.63e-16

 matrix columns 7 thru 10
 6.15e-17   2.93e-17   1.04e-17   1.45e-18
> rank(b)
        5
```

See also: Page 86 `epsilon()`, Page 142 `norm()`, Page 201 `size()`.

rcond — Condition Number

RCond



`rcond` calculates a scalar that is an approximation to the reciprocal of the condition number of the argument matrix. An approximation is used to reduce the computation required.

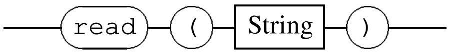
Example

```
> a = rand(6,6)
 a =
 matrix columns 1 thru 6
    0.707      0.927      0.558      0.537      0.299      0.411
    0.798      0.316      0.108      0.699      0.703      0.356
    0.366     0.0293      0.207      0.589      0.485       0.65
     0.67      0.188     0.0663      0.431       0.43      0.443
    0.235      0.841      0.959       0.12      0.536      0.233
    0.573        0.8      0.769      0.113      0.365      0.336

> norm(a)*norm(inv(a))
      185
> rcond(a)
   0.0054
> 1/rcond(a)
      185
```

See also: Page 77 `det()`, Page 119 `inv()`, Page 130 `lu()`.

read — File Read

Read



`read` is a function that allows you to read in variables from a file. The string argument specifies the file to read from. All the variables that are stored in the file are read in, overwriting any existing variables with the same name. The file is closed after it has been read from. This function returns 1 is the read succeeds.

The format of the file is quite specific, and it is intended that this function be used with the `write` function. Trying to create a file that can be `read` from is fraught with danger. If it is essential, I suggest using `write` to create one similar, and then editing as little as necessary.

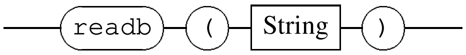
Example

```
Dummy Example
```

See also: Page 68 `close()`, Page 184 `readm()`, Page 233 `write()`, Page 236 `writem()`.

readb — Read binary data from a file

ReadMatrix



`readb` is used to read in binary data from a file.
This function is directly compatible with the MATLAB^{TM} matrix storage method.

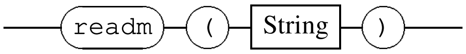
Example

```
Dummy Example
```

See also: Page 68 `close()`, Page 183 `readb()`, Page 184 `readm()`, Page 188 `reshape()`, Page 233 `write()`, Page 236 `writem()`.

readm — Read Matrix from file

ReadMatrix



`readm` is used to read in a matrix argument. The argument specifies the file to read it from.

The file format is generic ASCII. The rows of the matrix are separated by newlines, and the columns are separated by spaces or tabs. `readm` is intended to read in data from other programs, either directly or using a simple script. You shouldn't try to read in a string sequence using this function - the results will be strange.

This function is directly compatible with the MATLAB^{TM} matrix storage method.

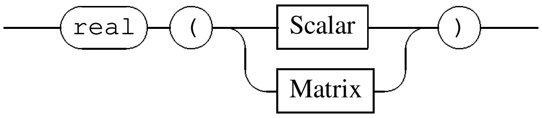
Example

```
Dummy Example
```

See also: Page 68 `close()`, Page 184 `readm()`, Page 188 `reshape()`, Page 233 `write()`, Page 236 `writem()`.

real — Real Part

Real



`real` returns the real part of the argument. If the argument is a matrix, then the operation is performed element-by-element, returning a matrix.

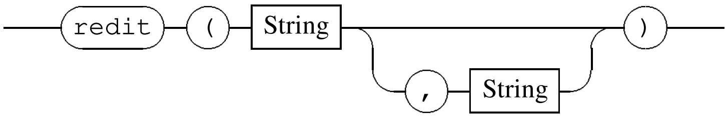
Example

```
> real(2+5j)
        2
> real(0.45+5.4326j)
     0.45
> real(3.32)
     3.32
> a = 100*(rand(2,3)+rand(2,3)*(0+1j))
 a =
 matrix columns 1 thru 3
             100 + 66.5i            64.7 + 20.4i            3.69 + 65.5i
            97.5 + 8.47i            33.3 + 16.7i            16.2 + 12.9i
> real(a)
      100       64.7       3.69
     97.5       33.3       16.2
```

See also: Page 71 `conj()`, Page 113 `imag()`.

redit — Edit rfiles

Redit



The `redit` function is a simple way to edit an rfile, and to then have it automatically reloaded upon exiting the editor. The first argument specified the name of the rfile to be editted, including the `.r` extension. The optional second string specifies the editor to use. The default editor is `vi(1)`[2]

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `redit.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `redit.r` may make this function unavailable.

Example

```
> redit("redit.r", "emacs");
edited and loaded rfile: redit.r
> system("diff redit.r redit.r.~1~");
28c28
<     ED = "emacs";
---
>     ED = "vi";
32c32
<       error ("2nd argument to edit() must be string");
---
>       error ("2nd argument ot edit() must be string");
```

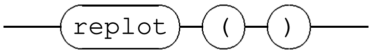
See also: Page 61 `cd()`, Page 218 `system()`.

---

[2] This decision is possibly the most perverse part of RLAB . Complaints to Ian Searle. — bradh

replot —

Replot



```
replot
```

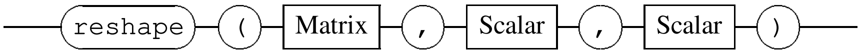
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

reshape — Reshape matrix

Reshape



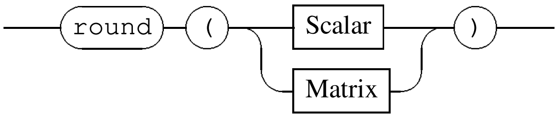
`reshape` changes the internal form of the argument matrix such that the matrix returned has the number of rows specified by the second argument, and the number of columns specified by the last argument. `reshape` will not reform the matrix if the new matrix and the old matrix would not have the same number of elements.

Example

```
> a = 1:20;
> show(a)
   name:      a
   class:     num
   type:      real
     nr:      1
     nc:      20
> b = reshape(a,4,5)
 b =
        1          5          9         13         17
        2          6         10         14         18
        3          7         11         15         19
        4          8         12         16         20
> c = reshape(b,2,10)
 c =
 matrix columns 1 thru 6
        1          3          5          7          9         11
        2          4          6          8         10         12

 matrix columns 7 thru 10
       13         15         17         19
       14         16         18         20
```

round — Round off value

Round



`round` returns the nearest integer value to the argument. If the argument is a matrix, the operation is done element by element.

The returned value is somewhat dependant on the underlying math library function, though in general, if the difference between the function argument and the rounded result is exactly 0.5, then the result will be rounded to the nearest even integer.

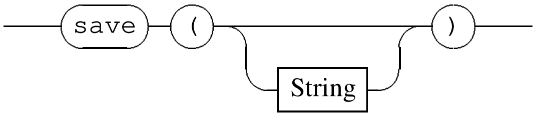
Example

```
> round(1.002)
        1
> round(1.998)
        2
> round(1.5)
        2
> round(2.5)
        2
> b = 100*rand(2,5)
 b =
     65.5         91       29.9         70       9.18
     12.9       11.2       26.5         95       90.2
> round(b)
       65         91         30         70          9
       13         11         27         95         90
> c = 100*(rand(2,3)+(rand(2,3)*(0+1j)))
 c =
 matrix columns 1 thru 3
              96 + 90.8i            44.1 + 14.8i           92.4 + 0.543i
            91.5 + 36.2i            7.35 + 87.9i            8.82 + 22.2i

> round(c)
 matrix columns 1 thru 3
                96 + 91i                44 + 15i                 92 + 1i
                91 + 36i                 7 + 88i                 9 + 22i
```

See also: Page 62 `ceil()`, Page 100 `floor()`, Page 116 `int()`.

save — Write workspace to a file

Save



The `save` function writes the contents of all the workspace variables to a file. The file is then closed. The file may be specified in the optional argument, with a default of `SAVE`. Functions are not saved. The variables can be read back from the file by using the `read` function (see Page 182).

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `save.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `save.r` may make this function unavailable.

Example

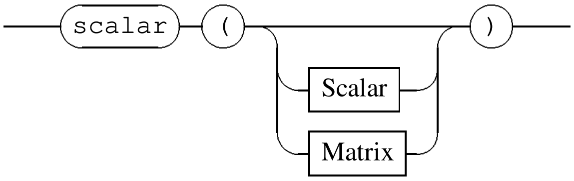
```
Dummy Example
```

See also: Page 68 `close()`, Page 182 `read()`, Page 233 `write()`.

]

scalar — Scalar Conversion

Scalar



`scalar` converts its argument to a scalar. If no argument is supplied, then 0 is returned. If the argument is a scalar, that scalar is returned. If the argument is a matrix, the matrix must have a single element, which is returned.

`scalar` is provided for symmetry with the `matrix` function. Functionally it is not required, since scalar references to matrices are automatically converted to scalar type.

Example

```
> scalar()
        0
> scalar(2)
        2
> scalar(2.3462+365.365j)
     2.35 + 365i
> scalar([1.3242e-21])
 1.32e-21
```

See also: Page 133 `matrix()`.

schord — Ordered Schur decomposition

Schord

```
——( schord )—( ( )—[ List ]—( , )—[ Scalar ]—( , )—[ Scalar ]—( ) )——
```

`schord` is a function used for post-processing the result of `schur`. The list argument is understood to be the return value from `schur`, and must contain elements `t` and `z`. The result of `schord` is a list containing elements `t` and `z`. It is possible to concatenate use of `schord`.

The basic idea of `schord` is to rearrange the `t` matrix to move the diagonal block in the row given by the first scalar argument to the row given by the second scalar argument.

Example

```
> root2on2 = sqrt(2)/2
 root2on2 =
    0.707
> M = [ root2on2, root2on2,0; root2on2, root2on2, 0;0,0,1]
 M =
    0.707       0.707           0
    0.707       0.707           0
        0           0           1
> R = schur(M)
 R =
   t               z
> new=  schord(R,1,3)
 new =
   t               z
> new.t
-1.11e-16           0           0
        0           1           0
        0           0        1.41
> new.z
   -0.707           0       0.707
    0.707           0       0.707
        0           1           0
> new.z*new.t*new.z'
```

`variable, or modifying set.r may make this function` unavailable.

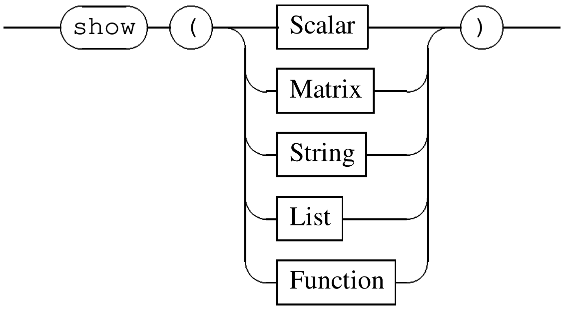
Example

```
> a = [1:5,-3.5,-1,6,87,3,9,0.-2]
 a =
 matrix columns 1 thru 6
        1           2           3           4           5         3.5
 matrix columns 7 thru 12
        1           6          87           3           9          -2
> set (a)
 matrix columns 1 thru 6
       -2           1           2           3         3.5           4
 matrix columns 7 thru 10
        5           6           9          87
```

See also: Page 70 `complement()`, Page 118 `intersection()`, Page 228 `union()`.

show — Display Characteristics

Show



`show` takes a single argument and returns a brief description of the argument, containing its class, name, and any other pertinent characteristics.

All the information provided by show() can be obtained by other means, such as direct member reference or `class` and `type` functions.

The information returned varies according to the argument. For scalars, strings and matricies, the result is of the same form. It is a five element string row vector with these elements:

`name` this is the name of the argument, or `NULL` if the argument is unamed.

`class` this is `num` for scalars and numeric matricies, and `string` for strings and string matricies.

`type` this is real or complex for scalars and numeric matricies, and blank for strings and string matricies.

`nr` this is `1` for scalars and strings, and the number of rows if the argument is a matrix.

`nc` this is `1` for scalars and strings, and the number of columns if the argument is a matrix.

For a list, there are three strings in the row vector, consisting of:

`name` the name of the list, or `NULL` if the list is unnamed.

`class` this is always `list`.

`n` this is the number of elements in the list.

There are also three elements to the result of this function with a function argument. They are:

`name` the name of the function.

`class` this is always `function`.

`type` this is `user` for user defined functions, and `builtin` for RLAB built in functions.

Example

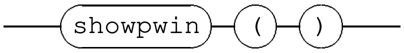
```
> a = 3.5+2j
 a =
      3.5 + 2i
> show(a)
   name:      a
   class:     num
   type:      complex
     nr:      1
     nc:      1
```

```
> b = [2,3,4;-5,-6,7]
 b =
        2          3          4
       -5         -6          7
> show(b)
   name:      b
   class:     num
   type:      real
     nr:      2
     nc:      3
> show("a string")
   name:      NULL
   class:     string
    type:     string
     nr:      1
     nc:      1
> c = "another string"
 c =
another string
> show(c)
   name:      c
   class:     string
    type:     string
     nr:      1
     nc:      1
> d = <<number=a; mat6=b;c>>
 d =
   3            mat6         number
> show(d)
   name:      d
   class:     list
       n:     3
> show(sin)
   name:      sin
   class:     function
    type:     builtin
> show(acosh)
   name:      acosh
   class:     function
    type:     user
```

See also: Page 65 `class()`, Page 227 `type()`.

showpwin — Current plot status

ShowPWin



`showpwin` reports the current status of the plotting system. It returns the current plot number, and the available sub-plots.

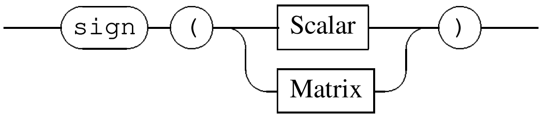
Example

```
No example yet - coming soon
```

See also: Page 1 `()`,

sign — sign of the argument

Sign



`sign` returns the sign of the argument. If the argument is greater than zero, it returns 1. If the argument is less than zero, it returns -1. Otherwise zero is returned. If the argument is a matrix, the operation is performed element-by-element, returning a matrix of the same dimensions as the argument. If the argument is complex, then `sign` returns the argument divided by its magnitude.

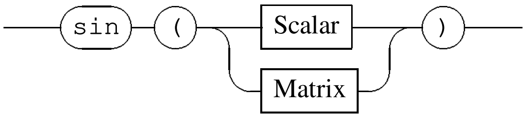
$\Longrightarrow$ This is not an RLaB built-in function. This function is normally loaded on start-up from the `sign.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `sign.r` may make this function unavailable.

Example

```
> sign(3)
        1
> sign(-3)
       -1
> sign(0)
        0
> sign(3+4j)
      0.6 + 0.8i
> a = (rand(2,5)-0.5)*10
 a =
        5       1.47      -4.63       1.65      -2.96
     4.75      -1.67      -3.38      -4.15      -3.33
> sign(a)
        1          1         -1          1         -1
        1         -1         -1         -1         -1
```

sin — Sine

Sine



`sin` calculates the sine of the argument, which must be specified in radians. If the argument is a matrix, then the `sin` operation is performed on each element of the matrix in turn. If the argument is complex, given by z, then the result is:

$$\sin(\Re(z))\cosh(\Im(z)) + j\cos(\Re(z))\sinh(\Im(z))$$

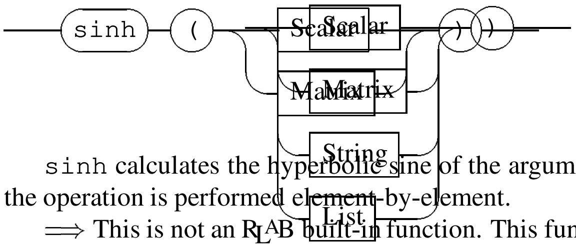
Example

```
> sin(0)
      0
> sin(pi/4)
  0.707
> sin(-pi/2)
     -1
> a = rand(2,5)*pi
 a =
  0.641       2.06       2.86      0.939        2.2
  0.526      0.405      0.351      0.834       2.99
> sin(a)
  0.598      0.884      0.279      0.807      0.809
  0.502      0.394      0.344      0.741      0.156
```

See also: Page 53 `asin()`, Page 72 `cos()`, Page 200 `sinh()`, Page 219 `tan()`.

sinh — Hyperbolic Sine

Sinh



`sinh` calculates the hyperbolic sine of the argument, specified in radians. If the argument is a matrix, then the operation is performed element-by-element.

⟹ This is not an RLaB built-in function. This function is normally

The `sizeof` function returns the number of bytes of data in the argument. This is somewhat machine dependent; though in general, the size of a:

real scalar is the same as a C double type.

complex scalar is twice as large as a real scalar.

matrix is equal to the number of elements multiplied by the size of the elements (real or complex).

string is equal to the number of characters.

list is equal to the sum of the sizes of the elements, assuming real scalars are promoted to complex scalars within the list.

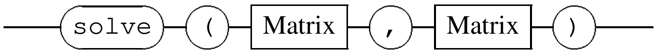
Example

```
> sizeof(2.432)
        8
> sizeof(1+3j)
       16
> a = rand(2,4)
 a =
        1      0.647     0.0369      0.665
    0.975      0.333      0.162     0.0847
> sizeof(a)
       64
> sizeof("1234567890")
       10
> str = "This string has 29 characters"
 str =
This string has 29 characters
> sizeof(str)
       29
> sizeof(<<2>>)
       16
> sizeof(<<3+3j>>)
       16
> sizeof(<<MatrixA = a>>)
       64
```

See also: Page 201 `size()`, Page 231 `who()`, Page 232 `whos()`.

solve — Linear Equation Solution

Solve



`solve` is a function to find the solution to sets of linear equations. If the linear equation is of the form:

$$Ax = B$$

then A is the first argument to solve, and is the square, non-singular, co-efficients matrix. Each column in the second argument is used in turn as the constant part, B, and the each column in the resulting matrix corresponds to using the same column of B as the right hand side.

The idea behind this is that you may have a several sets of linear equations using the same co-efficients with each variable, but equated to different constants. Since most of the work is associated with the variable co-efficients, it adds little overhead to solve for several sets of constants, at the same time. Not withstanding this, most of the time both the second argument and the result will be column vectors, representing a set of simultaneous solutions.

Example

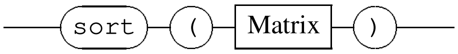
```
> A = rand(5,5)
 A =
   0.0882    0.00543      0.246      0.161      0.555
    0.908      0.222      0.782      0.975       0.29
    0.362       0.98      0.341     0.0106      0.205
    0.148       0.83      0.461      0.414      0.561
    0.879    0.00526       0.52      0.233      0.791
> X = rand(5,1)
 X =
    0.269
    0.334
    0.278
    0.789
    0.692
> B = A*X
 B =
    0.605
     1.51
     0.67
     1.16
     1.11
> //Now we should be able to use A and B to get X back
> soln = solve(A,B)
 soln =
    0.269
    0.334
    0.278
    0.789
    0.692
> // Check how close we are:
> soln - X
-2.22e-16
-1.67e-16
```

```
 8.88e-16
-4.44e-16
-2.22e-16
```

See also: Page 58 `backsub()`, Page 92 `factor()`, Page 119 `inv()`.

sort — Sort Matrix

Sort



`sort` has two different forms, depending on its argument. If the argument is a row or column vector, a list containing two vectors is returned. Both vectors will be the same size as the argument. One vector will be named `val`, and will contain the same elements as the argument, but sorted in ascending magnitude. The other vector is named `ind`, and tells you where the `val` element associated with it was in the argument. If the argument is a string vector, then the sorting is done in alphabetical order.

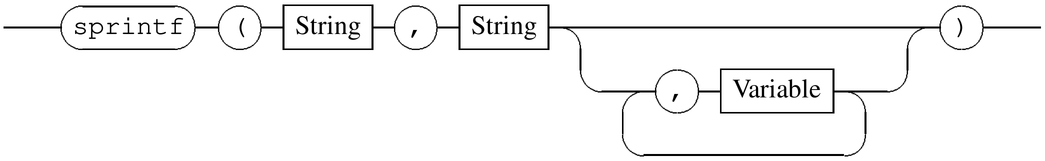
If the argument is not a vector, then the same thing happens, except that each column of the argument is sorted in turn.

Example

```
Dummy Example
```

sprintf — Create formatted string

FormattedString



The RLAB `sprintf` function allows a formatted string to be copied to a variable. It is similar to the C language function of the same name, though some features are limited or not available, since RLAB doesn't have all the data types of C.

The first argument is the name of the target string. If it is already a string, the old contents are deleted. If it is not already a string, then it is deleted, and a new string is created. If it doesn't exist, it is created.

The second string argument is the format string. It consists of the text to be written out, and possibly some conversion specifications. A conversion specification is a sequence of commands that determine how the remainder of the arguments are to be displayed. The left most conversion specifier is matched to the third argument, the next specifier to the fourth argument, and so on. Each argument has to be matched to a conversion specification. When the format string is being scanned, the argument that matches the the specifier that the scanner is up to is called the *current argument* .

Conversion specifications always begin with a `%` sign. The next thing that can occur are the flags:

- causing the conversion to be left-justified, instead of the default right-justified.

+ causing a sign to always be prepended to the conversion, instead of the default of only prepending minus signs.

which causes the format to be of an alternate form. This only has meaning for `e`, `E`, `f`, `g` and `G` formats, where is causes a decimal point to be used always. It also prevents suppression of trailing zeros for `g` and `G` options.

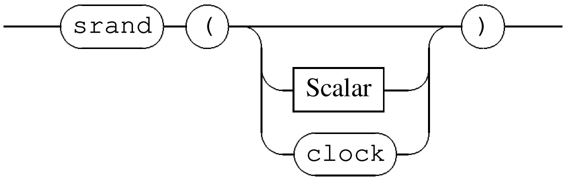
0 (zero) which causes padding by leading zeros, instead of the default space padding. This is overridden by both − and specifying a precision.

space which causes a space to be prepended if no sign is present. This is overridden by the + flag.

Following any flags, a minimum field width may be specified. This is either a integer constant, or a `*` character. If it is a constant, this is the minimum width. If it is a `*`, then the current argument (which must be a scalar) is used to specify the minimum width. If there is an optional width, then there may also

srand — Random Seed

RandomSeed



`srand` is used to set the seed for the random number generator routines in `rand`. With no arguments, `srand` resets the seed value to the last value which was set, or to 1, which is the seed value on startup, if it hasn't been set since. If a scalar argument is provided, it becomes the new seed. If the string `"clock"` is used as the argument, the seed is set from the internal clock. The latter method will provide unique values.

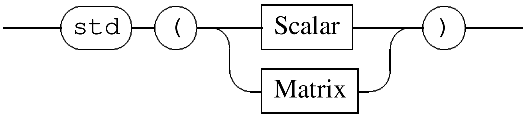
Example

```
> rand(2,4)
        1       0.647      0.0369       0.665
    0.975       0.333       0.162      0.0847
> rand(2,4)
    0.204       0.655        0.91       0.299
    0.167       0.129       0.112       0.265
> srand(1);
> // Reset the seed.
> rand(2,4)
        1       0.647      0.0369       0.665
    0.975       0.333       0.162      0.0847
> srand("clock")
        1
> rand(2,4)
    0.366       0.803        0.34      0.0192
    0.372        0.51       0.362       0.575
```

See also: Page 178 `rand()`.

std — standard deviation

StdDev



`std` calculates the standard deviation of the elements in the argument matrix. If the argument is a row or column vector, then the standard deviation of all elements is returned. Otherwise `std` returns a row-matrix formed by taking the standard deviation of the elements in each column.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `mst.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `std.r` may make this function unavailable.

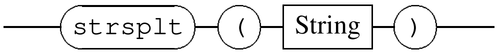
Example

```
> rand("normal", 10.0, 2.0 );
> a = rand(1,100);
> mean(a)
     9.76
> std(a)
     2.04
> b = rand(1,10)
 b =
 matrix columns 1 thru 6
     8.35       7.86       5.48       7.05       12.3       10.3

 matrix columns 7 thru 10
     13.1       6.73       11.4       7.18
> mean(b)
     8.96
> std(b)
      2.6
```

See also: Page 136 `mean()`.

strsplt — Split a string

Strsplt



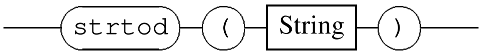
`strsplt` takes a character string as input, and returns a row matrix that contains a single character string as each element. You will get an error if the argument is not a string. String matricies are not allowed as input to `strsplt`. The resulting matrix has as many columns as the argument had characters.

Example

```
> string1 = "a string"
a string
> StringMatrix = strsplt(string1)
 StringMatrix =
a     s  t  r  i  n  g
> show(StringMatrix)
   name:      StringMatrix
   class:     string
   type:      string
     nr:      1
     nc:      8
> strsplt("")
        []
> show( strsplt(""))
   name:      NULL
   class:     string
   type:      string
     nr:      0
     nc:      0
> strsplt(25.5 + 4j)
rlab: strsplt, NULL, wrong type of arg to built-in should be: STRING
> strsplt("25.5 + 4j")
2  5  .  5     +     4  j
```

strtod — String to decimal conversion

StrToD



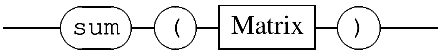
`strtod` takes a string argument, and returns the contents as a scalar value. Although this function understands exponentional notation, it does not understand complex notation.

Example

```
> strtod("1.23")
     1.23
> a = strtod("1.23")
 a =
     1.23
> class(a)
num
> b = strtod("2.12345e45")
 b =
 2.12e+45
> class(b)
num
> c = strtod("3+4j")
 c =
        3
> d = strtod("hello")
 d =
        0
```

sum — Summation of elements

Sum



`sum` computes the sum of a matrix. If the argument is a row or column vector, then the sum of all the elements in the vector is returned. Otherwise, the return object is a row matrix which contains the sum of each column of the input.

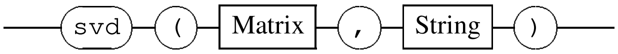
Example

```
> a = 1:3:0.5
 a =
        1        1.5            2        2.5            3
> sum(a)
       10
> b = (2:4)'
 b =
        2
        3
        4
> sum(b)
        9
> c = rand(5,3)*100
 c =
     26.5         96       8.82
       70       91.5       90.8
       95       44.1       36.2
     9.18       7.35       14.8
     90.2       92.4       87.9
> sum(c)
      291        331        239
```

See also: Page 76 `cumsum()`, Page 173 `prod()`.

svd — Singular Value Decomposition

SVD



`svd` calculates the singular values, and different forms of the left and right singular vectors. The result is returned as three elements in a list. If we are performing singular value decomposition of A, then

$$A = UI\sigma V^T$$

where U is the right singular vector, returned in `u`, σ is a vector of singular values, returned in `sigma`, and V^T is the left singular value, returned in `vt`.

Full singular values area always returned. However the form of the left and right singular vectors is dependant on the value of the optional string argument:

"A" or "a" The full U, and V^T are returned

"S" or "s" A minimal version of U, and V^T are returned. This is the default.

"N" or "n" U and V^T are not computed, empty `u` and `vt` elements are returned.

Full singular vectors means that the null-space is represented in the singular values. This is the slowest option. Minimal singular vectors means that sufficient U and V^T are returned to reproduce A. In general, for an argument of dimensions rxc, then if $r \geq c$,

Format	U	σ	V^T
Full	rxr	$1xc$	cxc
Minimal	rxc	$1xc$	cxc
Nil	$0x0$	$1xc$	$0x0$

For $r \leq c$, then

Format	U	σ	V^T
Full	rxr	$1xr$	cxc
Minimal	rxr	$1xr$	rxc
Nil	$0x0$	$1xr$	$0x0$

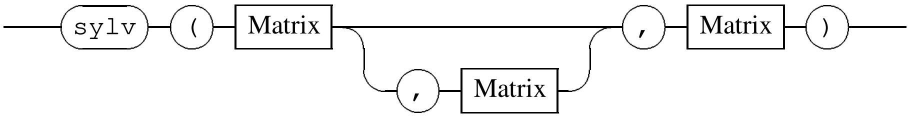
Example

```
> A = [0.96, 1.72; 2.28, 0.96];
> Asvd = svd(A)
   sigma        u            vt
> Asvd.vt
 matrix columns 1 thru 2
        -0.8        -0.6
         0.6        -0.8
> Asvd.u
 matrix columns 1 thru 2
        -0.6        -0.8
        -0.8         0.6
> Asvd.sigma
 vector elements 1 thru 2
           3           1
> check = Asvd.u * diag(Asvd.sigma) * Asvd.vt
 check =
 matrix columns 1 thru 2
```

```
        0.96        1.72
        2.28        0.96
> a = rand(100,20);
> b = svd(a)
 b =
   sigma        u            vt
> whos(b)
        Name            Class   Type    Size            NBytes
        sigma           num     real    1       20      160
        u               num     real    100     20      16000
        vt              num     real    20      20      3200
Total MBytes = 0.019360
        0
> c = svd(a,"A")
 c =
   sigma        u            vt
> whos(c)
        Name            Class   Type    Size            NBytes
        sigma           num     real    1       20      160
        u               num     real    100     100     80000
        vt              num     real    20      20      3200
Total MBytes = 0.083360
        0
> d = svd(a,"N")
 d =
   sigma        u            vt
> whos(d)
        Name            Class   Type    Size            NBytes
        sigma           num     real    1       20      160
        u               num     real    0       0       0
        vt              num     real    0       0       0
Total MBytes = 0.000160
        0
```

sylv — solution of Sylvester Equation

Sylv



`sylv` is a Sylvester equation solver. Given the first matrix argument is A, the second matrix argument is B, and the third matrix argument is C, `sylv` solves

$$Ax + xB = -C$$

for x. If there are only two arguments, it is assumed $B = A^T$, and the Lyapunov equation is solved.

The matrix arguments must both be upper triangular. If they are not, the `lyap` function implements a wrapper over this function that performs suitable Schur decomposition.

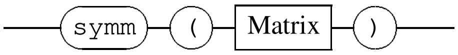
Example

```
> a = triu(rand(4,4))
 a =
    0.807      0.253      0.216      0.762
        0      0.339     0.0222      0.717
        0          0      0.356     0.0606
        0          0          0       0.78
> b = triu(rand(4,4))
 b =
    0.324      0.156       0.37      0.743
        0      0.133       0.81      0.705
        0          0      0.909      0.621
        0          0          0      0.845
> c = diag(rand(1,4))
 c =
    0.845          0          0          0
        0      0.401          0          0
        0          0      0.341          0
        0          0          0      0.623
> x = sylv(a,b,c)
 x =
   -0.747      0.353    -0.0535      0.294
       -0     -0.849      0.556      0.443
       -0         -0      -0.27      0.159
       -0         -0         -0     -0.383
> a*x+x*b + c
-1.11e-16  -5.55e-17          0  -5.55e-17
        0          0   5.55e-17          0
        0          0          0          0
        0          0          0          0
```

See also: Page 131 `lyap()`, Page 193 `schur()`.

symm — Symmetric Matrix

Symmetric



`symm` returns the symmetric part of the argument matrix. This is the nearest symmetric (Hermitian) matrix to the argument in both the Frobenius norm and 2-norm.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `symm.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `symm.r` may make this function unavailable.

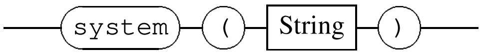
Example

```
> a = rand(3,3)+rand(3,3)*1j
 a =
 matrix columns 1 thru 3
          0.362 + 0.782i        0.00543 + 0.52i            0.83 + 0.0106i
          0.148 + 0.341i         0.222 + 0.161i         0.00526 + 0.414i
          0.879 + 0.461i          0.98 + 0.975i           0.246 + 0.233i

> symm(a)
 matrix columns 1 thru 3
              0.362 + 0i       0.0766 + 0.0896i           0.855 - 0.225i
        0.0766 - 0.0896i             0.222 + 0i            0.493 - 0.28i
          0.855 + 0.225i          0.493 + 0.28i               0.246 + 0i

> norm(a)
     2.23
> norm(a,"2")
     2.08
> norm(a,"f")
     2.26
> norm(symm(a))
      1.7
> norm(symm(a),"2")
     1.38
> norm(symm(a),"f")
     1.57
```

See also: Page 123 `issymm()`.

system — Access operating system

System



The `system` function allows RLaB programs to access the operating system. The string argument is passed directly to the default command interpreter for execution - normally the Bourne shell or some variation. RLaB waits until the system call is finished, and then returns you to the prompt.

The return value for this function is heavily system specific. Portable scripts should not use the return value from `system`.

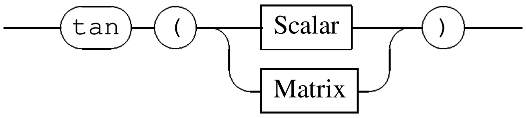
Example

```
> system("df")
Filesystem         1024-blocks  Used Available Capacity Mounted on
/dev/hdb5             250559  178079    59953     75%   /
/dev/hda1             207240  194580    12660     94%   /dosC
        0
> system("du -s");
2953    .
> system("bash");
bash$ echo "Now we have a shell running inside RLaB"
Now we have a shell running inside RLaB
bash$ w
  5:32am  up  8:09,  2 users,  load average: 0.03, 0.11, 0.09
User     tty      from             login@  idle   JCPU   PCPU  what
bradh    tty1                      9:24pm  8:08   1:05         (startx)
bradh    ttyp1                     9:53pm           22         w
bash$ exit
exit
> system("ls -l Makefile");
-rw-r--r--   1 bradh    bin          3537 Mar  6 00:33 Makefile
```

See also: Page 61 `cd()`, Page 108 `getline()`.

tan — Tangent

Tangent



`tan` returns the tangent of the argument, taken as being specified in radians. If the argument is a matrix, then the `tan` operation is performed on each element of the matrix in turn. If the argument is complex, the result is the ratio of the complex sine and complex cosine of the argument.

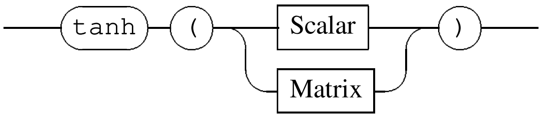
Example

```
> tan(0)
        0
> tan(pi/4)
        1
> tan(pi/2)
1.63e+16
> tan(-pi/6)
   -0.577
> a = rand(2,4)*pi
 a =
     3.14       2.03      0.116        2.09
     3.06       1.05      0.508       0.266
> tan(a)
-1.12e-06         -2      0.117       -1.76
  -0.0802       1.73      0.557       0.272
> tan(0+1j)
        0 + 0.762i
> tan(1+1j)
    0.272 + 1.08i
```

See also: Page 55 `atan()`, Page 72 `cos()`, Page 199 `sin()`, Page 220 `tanh()`.

tanh — Hyperbolic Tangent

Tanh



`tanh` returns the hyperbolic tangent of the argument, which is taken to be specified in radians. If the argument is a matrix, then the operation is performed element-by-element.

⟹ This is not an RLaB built-in function. This function is normally loaded on start-up from the `tanh.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `tanh.r` may make this function unavailable.

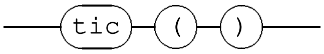
Example

```
> tanh(0)
       0
> tanh(1)
   0.762
> a = rand(2,5)
 a =
    0.91      0.299        0.7     0.0918       0.96
   0.112      0.265       0.95      0.902      0.915
> tanh(a)
   0.721       0.29      0.604     0.0916      0.744
   0.111      0.259       0.74      0.717      0.724
> tanh(0+1j)
      0 + 1.56i
> tanh(2+5.23j)
    1.02 - 0.0321i
```

See also: Page 50 `acosh()`, Page 54 `asinh()`, Page 57 `atanh()`, Page 73 `cosh()`, Page 90 `exp()`, Page 200 `sinh()`, Page 219 `tan()`.

tic — Start Timer

Timer



`tic` is used to start and reset the timer. It internally marks the time at which it was invoked. To measure elapsed time, use this function in conjunction with `toc`.

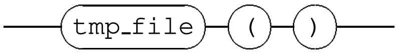
Example

```
> tic();
> toc()
        3
> // A bit bigger delay this time:
> toc()
       23
> // Leave it for quite a long time:
> toc()
 2.08e+03
> 2080/60
     34.7
> // yep: that was nearly 35 minutes.
```

See also: Page 223 `toc()`.

tmp_file — unique filename generator

TmpFile



The `tmp_file` function generates a string which can be used as a temporary file name. The resulting string starts with `rlab-`, then has a time (of the form *hhmmss*), ending with as a unique number.

It is possible, if unlikely, that this routine can generate two identical filenames. If you can generate a filename, reload `tmp_file.r`, and generate another filename, all inside one second, then they will be the same. This is so unlikely as to be perverse. More likely is that generating a filename at the same time each day will produce the same filename.

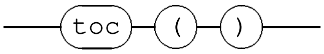
⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `tmp_file.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `tmp_file.r` may make this function unavailable.

Example

```
> tmp_file()
rlab-181200.1
> tmp_file()
rlab-181210.2
> tmp_file()
rlab-181218.3
> tmp_file()
rlab-181247.4
> load("/usr/local/lib/rlab/rlib/tmp_file.r"); a = tmp_file() ...
> load("/usr/local/lib/rlab/rlib/tmp_file.r"); b = tmp_file()
 a =
rlab-181524.1
 b =
rlab-181524.1
```

toc — Read timer

ReadTimer



`toc` measures the time that has elapsed since the last call to the `tic` function. It returns a scalar containing the time difference, measured in seconds. If `tic` has not previously been called, an error will occur.

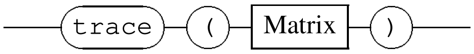
Example

```
> toc()
rlab: must call tic() 1st
> tic();
> toc()
         11
> tic(); a = rand(20,200); toc()
          2
> tic(); a = rand(200,200); toc()
         16
> tic(); b = inv(a); toc()
  2.39e+03
> 2390/60
       39.8
> // Yep, that took nearly 40 minutes.  Now, why did I want that ...
```

See also: Page 221 `tic()`.

trace — trace of a matrix

Trace



`trace` returns the trace of the argument matrix. This is the sum of the terms on the main diagonal.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `trace.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `trace.r` may make this function unavailable.

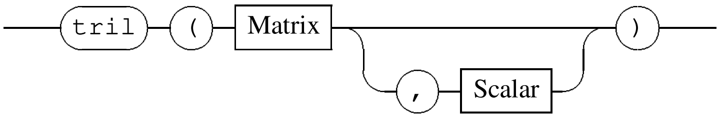
Example

```
> a = rand(3,3)
 a =
        1      0.333      0.665
    0.975     0.0369     0.0847
    0.647      0.162      0.204
> trace(a)
     1.24
> 1+0.0369+0.204
     1.24
> trace(eye(5,5))
        5
```

See also: Page 213 `sum()`.

tril — lower triangular matrix

Tril



`tril` returns the lower triangular part of the argument matrix. If no scalar argument is present, this is all the terms on the main diagonal and below. The remainder of the matrix is set to zero. If the optional scalar argument is present, then it specifies the diagonal to use instead of the main diagonal. The convention used is that positive values are above the main diagonal, negative values are below the main diagonal, and zero is the main diagonal.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `tril.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `tril.r` may make this function unavailable.

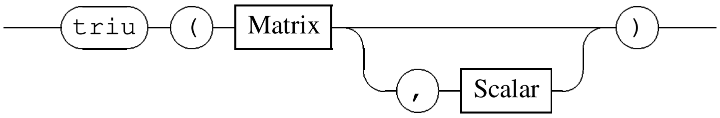
Example

```
> b = rand(3,5)
 b =
         1      0.333      0.665      0.167       0.91
     0.975     0.0369     0.0847      0.655      0.112
     0.647      0.162      0.204      0.129      0.299
> tril(b)
         1          0          0          0          0
     0.975     0.0369          0          0          0
     0.647      0.162      0.204          0          0
> tril(b,1)
         1      0.333          0          0          0
     0.975     0.0369     0.0847          0          0
     0.647      0.162      0.204      0.129          0
> tril(b,3)
         1      0.333      0.665      0.167          0
     0.975     0.0369     0.0847      0.655      0.112
     0.647      0.162      0.204      0.129      0.299
> tril(b,-1)
         0          0          0          0          0
     0.975          0          0          0          0
     0.647      0.162          0          0          0
```

See also: Page 226 `triu()`.

triu — upper triangular matrix

Triu



`triu` returns the upper triangular part of the argument matrix. If no scalar argument is present, this is all the terms on the main diagonal and above. The remainder of the matrix is set to zero. If the optional scalar argument is present, then it specifies the diagonal to use instead of the main diagonal. The convention used is that positive values are above the main diagonal, negative values are below the main diagonal, and zero is the main diagonal.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `triu.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `triu.r` may make this function unavailable.

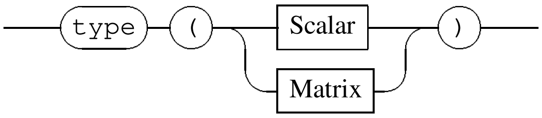
Example

```
> c = rand(3,5)
 c =
        1      0.333      0.665      0.167       0.91
    0.975     0.0369     0.0847      0.655      0.112
    0.647      0.162      0.204      0.129      0.299
> triu(c)
        1      0.333      0.665      0.167       0.91
        0     0.0369     0.0847      0.655      0.112
        0          0      0.204      0.129      0.299
> triu(c,2)
        0          0      0.665      0.167       0.91
        0          0          0      0.655      0.112
        0          0          0          0      0.299
> triu(c,-1)
        1      0.333      0.665      0.167       0.91
    0.975     0.0369     0.0847      0.655      0.112
        0      0.162      0.204      0.129      0.299
```

See also: Page 225 `tril()`.

type — description of argument type

Type



`type` returns a string that describes the type of element contained in the argument. The results are:

real a real scalar or matrix

complex a complex scalar or matrix

string a string or string matrix

user a user defined function

builtin a built in function

In addition, if the argument is a list, with a string element named `type`, that string will be returned. If the list doesn't have an element named `type`, or that element doesn't contain a string, the return value is an empty string.

Example

```
> type(2.43)
real
> type(2.43+4j)
complex
> type("this is a string")
string
> type(sin)
builtin
> type(sin(3.14))
real
> type(union)
user
```

See also: Page 65 `class()`, Page 195 `show()`.

union — Union of two sets

Union

The `union` function returns a vector set that is the union of the two sets supplied as arguments. A set is just a row or column vector.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `union.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `union.r` may make this function unavailable.

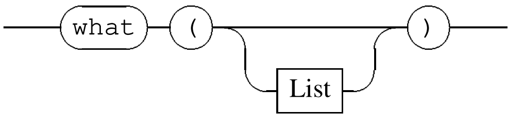
Example

```
> a = 1:3
 a =
        1          2           3
> b = [2,5,8,-2]
 b =
        2          5           8         -2
> union(a,b)
       -2          1           2          3          5           8
```

See also: Page 70 `complement()`, Page 118 `intersection()`, Page 194 `set()`.

what — List all functions

What



`what` without any arguments returns a string matrix of all the built-in and user-defined functions that currently reside in the global symbol table. If you supply a list argument to what, it will print out a list of the user-defined functions in the list.

Example

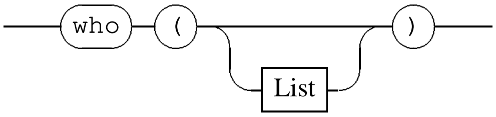
```
> what()
abs           eign          length        plot3         sizeof
acos          eigs          linspace      plprint       solve
acosh         epsilon       load          plptex        sort
all           error         log           plstyle       sprintf
any           eval          log10         plwid         sqrt
asin          exist         logspace      printf        srand
asinh         exp           lu            printmat      std
atan          eye           lyap          prod          strsplt
atan2         factor        matrix        pstart        strtod
atanh         fft           max           ptitle        sum
backsub       filter        maxi          pwin          svd
balance       find          mean          qr            sylv
cd            finite        members       rand          symm
ceil          fix           min           rank          system
chol          floor         mini          rcond         tan
class         format        mod           read          tanh
clear         fprintf       nan           readb         tic
clearall      fvscope       norm          readm         tmp_file
close         getenv        num2str       real          toc
compan        getline       ode           redit         trace
complement    hess          ones          replot        tril
conj          hilb          pause         reshape       triu
cos           ifft          pclose        round         type
cosh          imag          pend          save          union
cross         inf           plalt         scalar        what
cumprod       input         plaspect      schord        who
cumsum        int           plaxis        schur         whos
det           int2str       plaz          set           write
diag          intersection  plfont        show          writeb
diary         inv           plgrid        showpwin      writem
diff          isempty       plgrid3       sign          xlabel
disp          isinf         plhist        sin           ylabel
dot           isnan         plimits       sinh          zeros
eig           issymm        plot          size          zlabel
>  mysin = function ( x ) { return sin (x); }
        <user-function>
>  mycos = function ( x ) { return cos (x); }
        <user-function>
> funclist = << cosine = mycos; sine = mysin ; v = 1:4; s = 2.43+3j>>
```

```
   cosine        s              sine          v
> what(funclist)
cosine   sine
```

See also: Page 231 `who()`.

who — list all variables

Who



`who` has two forms. The first form takes no arguments, and returns a string matrix that lists all the variables currently in the global symbol table. The second form take a list argument, and returns a string matrix containg the names of all the elements in that list.

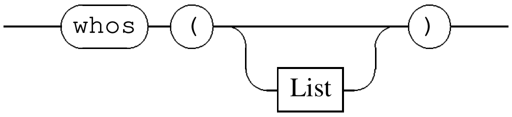
Example

```
> who()
eps  pi
> a = "a string"
a string
> b = 23.54
 b =
     23.5
> c = rand(2,5)
 c =
        1       0.647     0.0369      0.665       0.204
    0.975       0.333      0.162     0.0847       0.167
> L.string = a;
> L.TwentyThree = b;
> L.aMatrix = c;
> who()
L    b    eps
a    c    pi
> who(L)
TwentyThree  aMatrix      string
```

See also: Page 229 `what()`.

whos — list of variables

Whos



`whos` is a function that displays the variables in a list, and the total space those variables take up. If no argument is specified, or the argument is not a list, then the entire global symbol table is displayed. If a list argument is supplied, then only the variables in that list are displayed.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `whos.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `whos.r` may make this function unavailable.

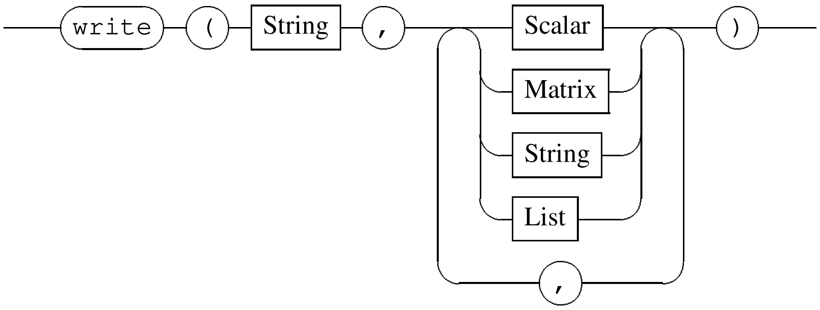
Example

```
> BigMatrix = rand(200,300);
> d = "a rather small string";
> whos();
        Name            Class   Type    Size            NBytes
        BigMatrix       num     real    200     300     480000
        d               string  string  1       1       21
        eps             num     real    1       1       16
        pi              num     real    1       1       16
Total MBytes = 0.566157
> L.a = "string for testing";
> L.B = "another string";
> L.c = rand(7,254);
> L.d = 3+4j;
> whos(L);
        Name            Class   Type    Size            NBytes
        B               string  string  1       1       14
        a               string  string  1       1       18
        c               num     real    7       254     14224
        d               num     complex 1       1       16
Total MBytes = 0.014272
```

See also: Page 229 `what()`, Page 231 `who()`.

write — file output

Write



`write` is a function that allows you to save variables to a file. The first argument specifies the file that you want to save the variable(s) to. Successive argument(s) specify the variables to save. If the file exists, it is cleared before anything is written to it. The file is left open so you can write more stuff out to it with more calls to `write`.

Example

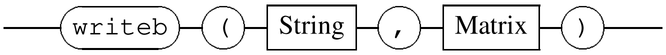
```
> matrix_name = rand(2,2)
 matrix_name =
        1      0.647
    0.975      0.333
> scal452 = 2.3456123
 scal452 =
     2.35
> scalC = pi + exp(1)*1j
 scalC =
     3.14 + 2.72i
> another_matrix = rand(3,1) + rand(3,1)*1j
 another_matrix =
        0.0369 + 0.0847i
          0.162 + 0.204i
          0.665 + 0.167i
> str = "some test text"
 str =
some test text
> DerList = << str; scalC; another_matrix >>
 DerList =
   1            2            3
> write("for_later", matrix_name);
> system("more for_later");
# matrix : matrix_name no_of_rows: 2 no_of_cols: 2 REAL
# matrix columns 1 thru 2
 0.9999996442347765       0.6474838890135288
 0.9745195945724845       0.3330855811946094
> write("_nother_file", scal452, matrix_name);
> system("more _nother_file");
# scalar : scal452
 2.3456123                0
# matrix : matrix_name no_of_rows: 2 no_of_cols: 2 REAL
```

```
# matrix columns 1 thru 2
 0.9999996442347765       0.6474838890135288
 0.9745195945724845       0.3330855811946094
> write("for_later", scalC, str, DerList, another_matrix);
> system("more for_later");
# matrix : matrix_name no_of_rows: 2 no_of_cols: 2 REAL
# matrix columns 1 thru 2
 0.9999996442347765       0.6474838890135288
 0.9745195945724845       0.3330855811946094
# scalar : scalC
 3.141592653589793        2.718281828459045
# string : str Length : 14
some test text
# list : DerList no_of_elements: 3
# string : 1 Length : 14
some test text
# scalar : 2
 3.141592653589793        2.718281828459045
# matrix : 3 no_of_rows: 3 no_of_cols: 1 COMPLEX
# matrix columns 1 thru 1
 0.03694454254582524      0.08467286918312311
 0.1617118660360575       0.2041363911703229
 0.6646450059488416       0.1673069256357849
# matrix : another_matrix no_of_rows: 3 no_of_cols: 1 COMPLEX
# matrix columns 1 thru 1
 0.03694454254582524      0.08467286918312311
 0.1617118660360575       0.2041363911703229
 0.6646450059488416       0.1673069256357849
> close("_nother_file");
> close("for_later");
> write("for_later", another_matrix);
> system("more for_later");
# matrix : another_matrix no_of_rows: 3 no_of_cols: 1 COMPLEX
# matrix columns 1 thru 1
 0.03694454254582524      0.08467286918312311
 0.1617118660360575       0.2041363911703229
 0.6646450059488416       0.1673069256357849
```

See also: Page 68 `close()`, Page 182 `read()`, Page 184 `readm()`, Page 236 `writem()`.

writeb — write binary data to file

WriteBinary



`writeb` is used to write binary data to a file. The output is intended to be compatible with both the `readb` function and several other programs, such as MATLAB^{TM}.

Example

```
Dummy Example
```

See also: Page 68 `close()`, Page 182 `read()`, Page 183 `readb()`, Page 184 `readm()`, Page 233 `write()`.

writem — write matrix to file

WriteMatrix

`writem` is used to write a matrix to a file. The output is intended to be compatible with both the `readm` function and several other programs, such as MATLAB^{TM}, and spreadsheets.

Each line in the output file contains all the values for the corresponding row in the matrix argument. The values within each line are delimited by whitespace. `writem` will output all kinds of matricies, but `readm` can only read numeric matricies with real values.

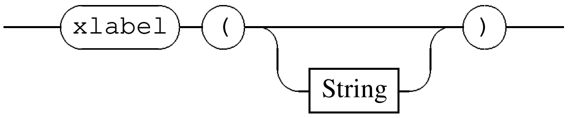
Example

```
Dummy Example
```

See also: Page 68 `close()`, Page 182 `read()`, Page 184 `readm()`, Page 233 `write()`.

xlabel — setting X axis labels

XLabel



`xlabel` sets the label used on the X axis of the current sub-plot. If this optional string argument is missing, or is a null length string, the label for the X axis is missing.

This routine must be called before doing the plot you want it to appear on.

The label is placed below the X axis, centred about the axis.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `plot.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `plot.r` may make this function unavailable.

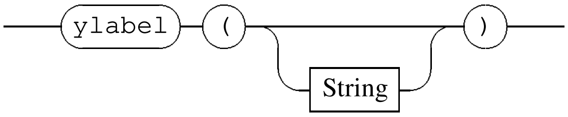
Example

```
> xlabel()
        0
> xlabel("A simple label")
        0
> xlabel("The independent variable - X");
```

See also: Page 238 `ylabel()`, Page 240 `zlabel()`, Chapter 8 Plotting.

ylabel — setting Y axis labels

YLabel



`ylabel` sets the label used on the Y axis of the current sub-plot. If this optional string argument is missing, or is a null length string, the label for the Y axis is missing.

This routine must be called before doing the plot you want it to appear on.

The label is printed sideways along the Y axis, centred on the axis.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `plot.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `plot.r` may make this function unavailable.

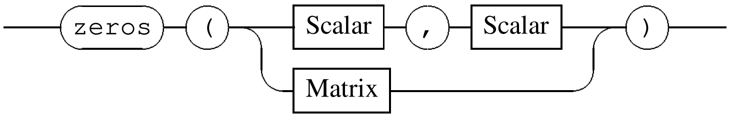
Example

```
> ylabel()
        0
> ylabel("Dependant Variable")
        0
> ylabel("This is Y");
```

See also: Page 237 `xlabel()`, Page 240 `zlabel()`, Chapter 8 Plotting.

zeros — matrix of zeros

zeros



`zeros` returns a matrix with all elements set to zero. If the arguments are two scalars, then the first argument specifies the number of rows, and the second argument specifies the number of columns. If the argument is a matrix, then `zeros` returns a matrix with the number or rows specified by the first argument, and number of columns specified by the second argument. This is the same form as the return value of `size`.

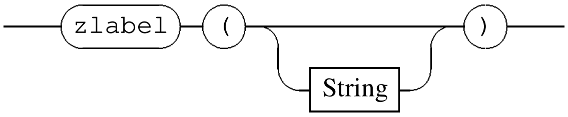
Example

```
> zeros(2,3)
        0          0          0
        0          0          0
> zeros([2,4])
        0          0          0          0
        0          0          0          0
> b = rand(3,5);
> size(b)
        3          5
> zeros(size(b))
        0          0          0          0          0
        0          0          0          0          0
        0          0          0          0          0
```

See also: Page 146 `ones()`, Page 201 `size()`.

zlabel — setting Z axis labels

ZLabel



`zlabel` sets the label used on the Z axis of the current sub-plot. If this optional string argument is missing, or is a null length string, the label for the Z axis is missing.

This routine must be called before doing the plot you want it to appear on.

This label is ignored if the current plot has only two dimensions.

⟹ This is not an RLAB built-in function. This function is normally loaded on start-up from the `plot.r` file in the standard `rlib` directory. Use of the `-r` option, incorrectly setting the `RLAB_LIB_DIR` environmental variable, or modifying `plot.r` may make this function unavailable.

Example

```
> zlabel()
        0
> zlabel("The third axis")
        0
> zlabel("Z axis");
```

See also: Page 237 `xlabel()`, Page 238 `ylabel()`, Chapter 8 Plotting.

Appendix A

Dis-assembling RLAB Programs

This chapter describes the purpose and usage of the RLAB disassembler, available by using the '`-d`' option from the command-line. The dis-assembler has several uses:

- Provides insight into the internal operation of the RLAB virtual machine.
- Provides an excellent tool for debugging new additions or changes to the operation of the virtual machine.

However, the dis-assembler will not normally help users debug their RLAB programs. It is a tool for working on the RLAB executable, rather than a tool for working with the RLAB language.

The dis-assembler produces a readable representation of the virtual-machine codes. It is recommended that the the '`-q`', '`-l`', and '`-n`' be used in conjunction with the '`-d`' option, since this combination will produce more readable output. It is almost essential, unless you are *extremely* patient, to use the '`-l`' option, which will prevent loading of all the R-files in the `rlib` directory. Failure to use '`-l`' in conjunction with '`-d`' will force you to wait an extremely long time while the op-codes for each library function are printed to the screen. The '`-n`' option merely suppresses the generation of file and line number information, and makes the output more readable.

For example, lets invoke RLAB with the correct options and look at the output from a simple expression:

```
augerinn:~$ rlab -qrldn
Welcome to RLaB. New users type `help INTRO'
RLaB version 0.97d beta Copyright (C) 1992, 93, 94 Ian Searle
RLaB comes
```

Index