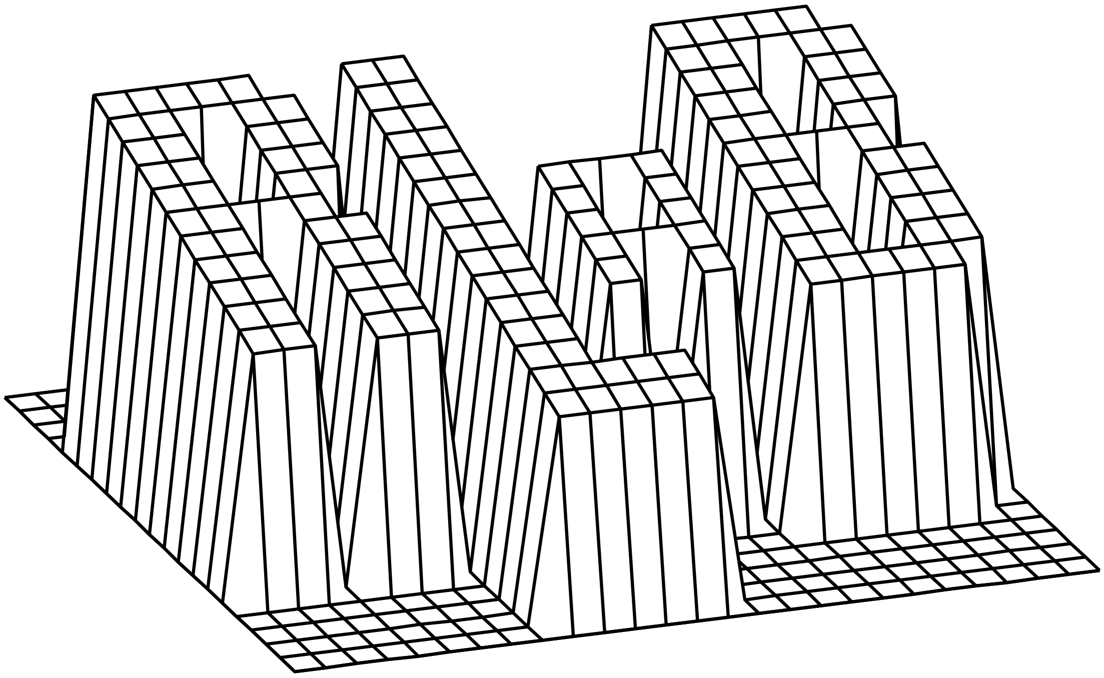
RLaB Primer

Version 1.0

Ian R. Searle & Phillip Musumeci



Contents

List of Figures

0 Primer priming

0.1 RLaB is freely available

RLaB stands for Our-Lab, since it is intended to be a freely available program that anyone can use, and contribute to. To protect this freedom, copying of the program is protected by the GNU General Public License.

The main ftp site is `evans.ee.adfa.oz.au`. The directory `pub/RLaB` contains the sources and binary versions for some machines. On the North American continent `csi.jpl.nasa.gov` acts as an archive site for RLaB, look in `pub/matlab/RLaB`.

0.2 Acknowledgments

The availability of "free" software, such as GNU Emacs, GNU gcc, gdb, gnuplot, Plplot, and the Netlib archives has made this project possible. The RLaB author thanks both the authors and sponsors of the GNU, LAPACK, RANLIB, FFTPACK, and Plplot projects.

Many individuals have contributed to RLaB in various ways. A list of contributors can be found in the source distribution file `ACKNOWLEDGMENT`. A special thanks to Phillip Musumeci and Matthew Wette who have provided FTP sites so that RLaB is available to all.

0.3 Document reproduction and errors

We welcome reports of errors and suggestions for improvement in this document and also in RLaB. Please mail these to `rlab-list@eskimo.com`. Unfortunately (for you), free software does not earn quite enough to pay a bribe for each error-free error report received but do feel free to email them.

0.4 Requirements

RLaB is written in C. The maths libraries used are written in Fortran but the use of a publicly available Fortran→C converter reduces compiler requirements to C (the conversion tool f2c is written in C). The library used for data display, PLPLOT, is publicly available in C source code form for a wide variety of platforms. This makes the whole RLaB package a good candidate for porting onto platforms with C, especially GNU C.

0.5 How to Read This Primer

This primer has intentionally been kept short, so you should be able to read all of it without too much effort. Probably the best way to read this primer is to do so sitting at a computer, trying the examples as you encounter them.

1 Introduction

RLaB brings the power of stable matrix maths tools plus a stable data plotting facility together in a form that is freely available and ready to be compiled and used on a variety of common computer systems. RLaB allows you to experiment with complex matrix maths in an interactive environment. Because you enter commands at a high (mathematical) level, you can concentrate on figuring out your solution and hopefully avoid becoming bogged down in low level implementation details. By minimising the effort required to implement algorithms, it is hoped that you will be more willing to discard old programs when confronted by better algorithms that warrant use.

RLaB uses a structured language[1] which will be familiar to users of C and also the Wirth-inspired languages such as Pascal and Modula. An RLaB program is a file containing a sequence of commands or instructions that you could also enter from your terminal—these instructions might perform a calculation and assign the result to a variable, or call a function which returns a result which you display on your terminal, and so on. Functions can be either built-in or user-defined. In fact, the only form of "subprogram" in RLaB is the function and, just like in C, a function returns a single item as its answer. Data storage declared in the main routine of your program is stored on a global symbol table, and is available to all of your subprogram functions. By default, data used within functions is local to the function. Such local function storage exists only for the duration of the function call, in a way similar to variables declared locally within Pascal procedures. Comments can be appended to any line in your program by using a special symbol at the start of the comment—this is similar to Fortran and C++, and avoids the possible pitfall of "run away" comments which might be familiar to Pascal users. Overall, the language syntax is perhaps closest to C but if you have ever programmed in C or Pascal, you will soon be at ease with RLaB .

RLaB features strongly typed objects but with the emphasis on usefulness, not on pedantics. In RLaB we talk about the *class* of an object and the available classes include numeric, string, function, and list. The first class of object, *numeric*, encompasses numeric scalars, vectors, and matrices, and should be familiar to the matrix maths user. The remaining classes borrow concepts, and implementation details from other languages such as C.

It is worth noting that a function can be thought of as just another object—this means that when you come to write your own functions that use input parameters, you will enjoy the flexibility of being able to pass in other functions as well as data as input to your function. Another feature of functions as implemented in RLaB is that they can call themselves—anyone who has written a program to calculate factorials will appreciate the elegance that recursion can bring to some programming solutions.

Having whetted your appetite, this primer aims to get you started with RLaB as both an interactive tool and as a programming language. The ideal approach is for you to read (or re-read) this document with an RLaB session staring up at you. After showing you how to run RLaB and get on-line help, we describe data types before moving back to a "hands on" description of basic operations. Program structure is then described and you will see how to write your own functions. As RLaB comes

[1] These days, computer languages do look very similar but we will try to point out a few *useful* similarities!

with quite a few handy functions already built-in, we give examples of their use including the plot function at which point we hope you will be able to start using RLaB to develop your own programs.

2 Starting to use RLaB

2.1 How to run it

A properly installed RLaB can be started on your terminal by entering

```
$ rlab
```

where typewriter-style dark text is meant to represent the text you would see sitting in front of a display terminal. The first character on the input line is always the prompt, in this case a Bourne-shell prompt. The text following is what the user enters. Text echoed by a program is not preceded by any prompt.

RLaB will start with a message similar to:

```
Welcome to RLaB. New users type 'help INTRO'
RLaB version 1.0 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
>
```

The `>` symbol on the last line next to the cursor is the RLaB command prompt. At this point, users should take the advice offered and be usefully distracted from this primer by *actually reading* the information available from `help INTRO` - do not worry if you cannot follow it yet. After you have read each screenful, press `SPACE` (i.e. the space bar) to see further screens of information.

At this point it is only fair to tell you how to stop it. To stop a RLaB session you can type `quit` at the RLaB prompt. On Unix systems an `EOF` or `^d` (control-d) will also stop RLaB .

2.2 Help

To get a taste of the functions for which help is available, enter

```
> help
```

The first group of topics lists functions and special help topics that are built into RLaB . The special topics have names in upper case and are of a general nature. Lawyers recommend that you *now* read the help on topics `CONDITIONS` and `WARRANTY` by entering

```
> help CONDITIONS
```

```
> help WARRANTY
```

The subsequent topics refer to commands that have been written in RLaB script files, which we refer to as "R-files". These R-files are stored in directories, which the `help` command searches. The help files in the .../*rlib* directory come as a standard part of RLaB, and the remainder refer to local R-files that have been setup for you by whoever installed your RLaB.

In general, the functions listed in the first group are the most efficient as they are compiled into the core of RLaB. In contrast, RLaB's R-files have the extra overhead of reading and interpretation before they are executed. This lower efficiency associated with R-file interpretation is traded for the benefit of being able to write your own features into RLaB. If an R-file feature is really useful, it can be added to the core RLaB program since you have the source code[2].

2.3 Simple calculations

RLaB is designed for mathematical calculations so let's do some. The four basic arithmetic operators have symbols `+, -, *, /` representing addition, subtraction, multiplication, and division respectively. Now enter some one line expressions as shown here:

```
> 2*4
        8
> 1/2
      0.5
> 1+11
       12
> 1-11
      -10
> 1*2/3+4-5
   -0.333
> 1/0
      inf
> 0/(1/0)
        0
> 0/0
      NaN
```

The 5^{th} expression illustrates compliance with the usual operator hierarchy and then we observe that RLaB can handle exceptions[3] such as when ∞ (`inf`) is a result or an input to further calculation; and also "not-a-number" (`NaN`). RLaB can use complex numbers as well as real numbers so now try

```
> 1/1i
        0 - 1i
```

[2]Dynamic linking is part of the plan for a future release

[3]The ability to properly handle floating point exceptions is dependent upon the capabilities of the hardware and operating system, as well as the interests of the person installing RLaB .

```
> 1/1i + 1/1j
        0 - 2i
> 1/1i * 1/1j
       -1
> 1/1i/1j
       -1
```

where we see that `i` or `j` can represent the complex number $\sqrt{-1}$. No four function calculator is complete without a memory so now we look at how to store results in a variable.

2.4 Variable assignment and display

In RLaB , variables can have names of any length containing most printable characters including $\{a \ldots z, A \ldots Z, \_\}$. You will observe that we have to exclude special characters such as `+`, `-`, `*`, `/` and the `SPACE` character. The actual assignment operator symbol is "=" and an assignment statement looks like

```
variable_name = expression_to_evaluate
```

and an example is

```
> radius=2
 radius =
        2
> circumference=2 * pi * radius
 circumference =
     12.6
```

where a variable `radius` is created and initialised with the real value 2, and then a variable `circumference` is created and filled with the result of evaluating the right hand side of the equation. To see the value of either of these variables, just enter their name and RLaB will print their value. For a description of variable names, please read the help on `VARIABLES`.

As you have probably noticed by now, the result of each expression is automatically printed to the screen. This feature can be controlled by using the '`;`' character. Terminating an expression with a '`;`' will suppress printing of the result. Likewise, terminating an expression with the '`?`' is an explicit way to force printing.

2.5 User Interface: command recall & editing

Command line recall and editing is very useful for correcting command errors or to allow your commands to evolve. RLaB provides a command recall and edit facility modeled on (and sometimes actually using) the GNU readline facility. If you are familiar with GNU emacs or the GNU bash shell, then try entering `C-p` to scroll back through previous commands (`C-p` means hold down the `control` key and press `p`). If this is successful, test the standard character and word editing commands to modify previous entries - if it works, skip to section 3.

However, if the word GNU just means "any of several African antelopes constituting the genus *Connochaetes* ..."[4] but your keyboard does have the arrow keys { $\leftarrow \uparrow \downarrow \rightarrow$ }, then you might still be able to take advantage of command line recall and editing. Try typing the $\uparrow$ key to see if any previous RLaB commands are displayed - if they are, then confirm that $\downarrow$ also displays more recent commands and then try horizontal cursor movement with the { $\leftarrow \rightarrow$ } and try some editing with the delete key. Typing `C-d` ought to delete the character beneath the cursor. When a new command has been created from an old, enter it in the usual way by pressing `RETURN`. If this has worked for you, skip the remainder of this section (and count yourself lucky that we weren't describing a graphical user interface in one paragraph).

If your keyboard is missing the arrow keys but `C-p` did cause previous commands to pop up on the RLaB command line, you will find that { $\leftarrow \uparrow \downarrow \rightarrow$ } are the same as { `C-b C-p C-n C-f` } - think of b for backwards, p for previous, n for next, and f for forward.

Irrespective of what keystrokes you use for editing, the C-y keystroke will restore text previously deleted. If you were unable to scroll back through any previous commands (that you had just entered), then your RLaB may have been built without command line editing - this is unlucky. As command line editing is such a useful feature, you should consider getting a better version of RLaB if possible.

3 Objects—Basic Data Structures

In the most general form, an object in RLaB can be data or a function. It is even possible to construct an object that contains both data *and* functions - a fact that no doubt excites the hormones in the modern day object oriented programmer. We are going to discuss basic data types before looking at how data can be "grouped" together for some useful purpose. We will also work through some simple examples that manipulate data but first, what does RLaB regard as data?

3.1 Data Types

There are three *fundamental* types of data that you manipulate in RLaB : the string; the real number; and the complex number. As we have seen in section 2.3, it is straight-forward to manipulate numerical quantities. Characters are available in the form of strings which can contain 0 or more characters. In line with a philosophy to "keep it simple", RLaB which is primarily concerned with numerical computation, has no special way to handle a single character[5]. To enter a string, enclose the characters inside quotes like `"this"` e.g.

```
> "Hello world"
Hello world
```

[4] Australian Macquarie dictionary, ISBN: 0-949757-23-3.

[5] And also one less data type that you need to learn about.

Just as a number was previously stored in a variable, the same can be done with a string of characters. To place a string into a variable, you could enter a statement such as

```
> hw = "Hello world"
Hello world
```

and the value of variable `hw` may be printed out by entering

```
> hw
Hello world
```

The observant reader might be wondering what has happened to the boolean data type? In RLaB , *true* and *false* are represented by the integers 1 and 0. Just as the data type char can be handled as a rather small string (length=1), so the data type boolean (or logical) can be handled by small numbers (value=0,1). We have now met the 3 fundamental types of data processed in RLaB and it is now possible to understand a little more about how data structures and functions are organised within RLaB .

3.2 Object Hierarchy

Scan your eyes down over Figure 1 which shows the hierarchical structure of objects in RLaB - we shall now describe this figure from the bottom up (ignoring lists until a little later). Not all objects are created in the same way and what you can do with or to them depends on their *class*. Items of class *function* contain program instructions which is one form of data or information. Items of class *numeric*, and *string* contain data that RLaB instructions can manipulate.

A numeric class item can store a real or complex number. An item of class string contains a null-terminated string of character(s). When we want to access or create an array of items, we use an array syntax that is the same for both string and numeric classes.

It is often helpful to a programmer to group together unlike data into a single object - this is the purpose of the class list. We are not going to describe it in great detail here except to point out that it serves a similar role to a record in Pascal or a structure in C, but with a somewhat more flexible access mechanism. Note that lists can contain any of the aforementioned objects, even another list.

One thing that you can always do with any item is ask RLaB what its class is. For example, RLaB has a built-in command to calculate the sin of an angular quantity - asking RLaB about it gives the following response

```
> class( sin )
function
```

From the size of the list of topics that help is available on, you probably realise that there are many built-in functions in RLaB - expect gratuitous use of these functions as further examples are given. Remember that you can find out about any function by typing `help` *function-name*. We are particularly interested in exploring the use of RLaB as a computation tool so now we describe further numeric operations.

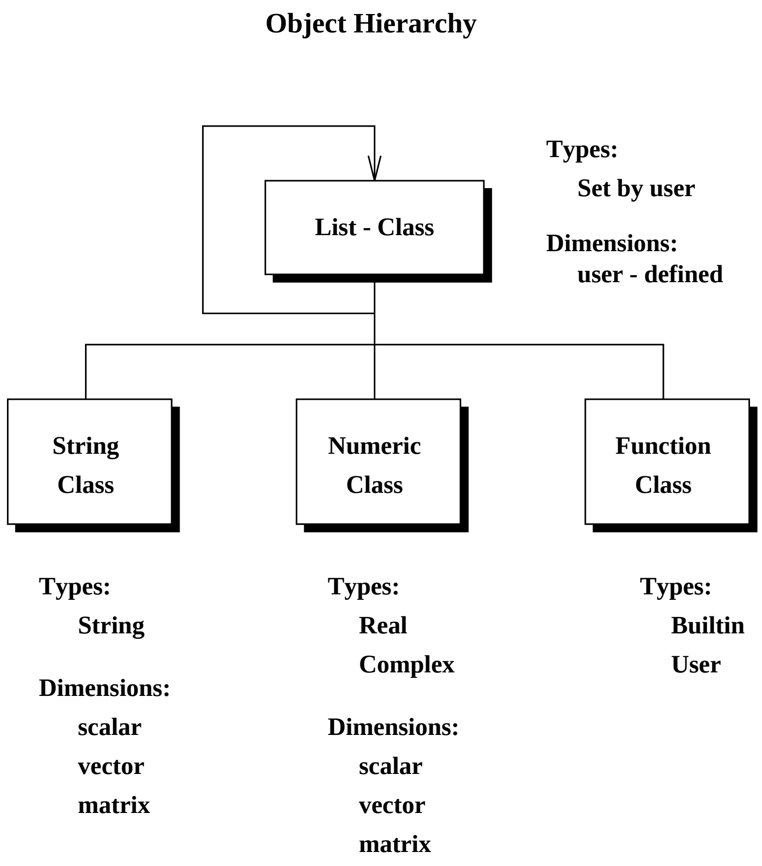


Figure 1: RLaB objects

3.3 Numerics

The RlaB numeric class includes objects of type real and complex. The numeric object also encompasses objects of scalar, vector, or matrix dimension. If you want to, you can think of all numeric objects as matrices. Thus, a vector is simply a 1-by-N matrix, and a scalar is a 1-by-1 matrix. Since the numeric object is most commonly used, it will get the most coverage.

3.3.1 Matrix Creation

The simplest way to create a matrix is to type it in at the command line:

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

In this context the '`[ ]`' signal RlaB that a matrix should be created. The inputs (or arguments) for matrix creation are whatever is inside the '`[ ]`'. The rows of the matrix are delimited with '`;`' and the elements of each row are delimited with '`,`'.

Users can use most any expression when creating matrix elements. Other matrices, function evaluations, and arithmetic operations are allowed when creating matrix elements. In the next example, we will create a direction cosine matrix using the built-in trigonometric functions within the '`[ ]`'.

```
> a = 45*(2*pi)/360
 a =
   0.785
> A = [ cos(a), sin(a); -sin(a), cos(a) ]
 A =
   0.707      0.707
  -0.707      0.707
```

Matrices can also be read from disk-files. The functions `read`, `readb` and `readm` can read matrix values from a file. The `read` function uses a special ASCII text file format, and is capable of reading not only matrices, but strings, and lists as well. Since the file can contain many data objects, and their variable names, `read` is used like:

```
> read ( "file.dat" );
```

The variables are read from `file.dat` and installed in the global-symbol-table.

The `readb` function works like `read`, except it reads binary files for greater efficiency. The binary files created with `writeb` are portable across computers that use IEEE floating point format.

The `readm` function reads a text file that contains white-space separated columns of numbers. `readm` is most often used to read in data created by other programs. Since `readm` is only capable of reading in one matrix per file, and no variable name information is available, `readm` is used like:

```
> a = read ( "a.dat" );
```

3.3.2 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 singly 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), that is '`start:end:inc`'. The leftmost operand, `start`, specifies the starting value, the second operand, `end`, specifies the last value. The default increment, or spacing, is 1. A third optional operand, `inc`, can be used to specify any increment.

```
> v = 1:4
 v =
        1           2           3           4
```

3.3.3 Matrix Attributes

Matrix attributes, such as number of rows, number of columns, total number of elements, are accessible in several ways. All attributes are accessible through function calls, for example:

```
> a = rand(3,5);
> show (a)
   name:      a
   class:     num
   type:      real
     nr:      3
     nc:      5
> size (a)
        3           5
> class (a)
num
> type (a)
real
```

Matrix attributes are also accessible via a shorthand notation:

```
> a.nr
        3
```

```
> a.nc
        5
> a.n
       15
> a.class
num
> a.type
real
```

Note that these matrix attributes are "read-only". In other words: assignment to `a.nr` is pointless. In fact it will destroy the contents of `a` and create a list with element named `nr`. If you wish to change a matrix attribute, you must do so by changing the data in `a`. For example: if you want to make `a` complex:

```
> a = a + zeros (size (a))*1i;
> show(a)
   name:      a
   class:     num
   type:      complex
     nr:      3
     nc:      5
```

If you want to change the number of rows, or columns of `a`:

```
> a = reshape (a, 1, 15);
> show(a)
   name:      a
   class:     num
   type:      complex
     nr:      1
     nc:      15
```

3.3.4 Element Referencing

Any expression that evaluates to a matrix can have its elements referenced. The simplest case occurs when a matrix has been created and assigned to a variable. One can reference single elements, or one can reference full or partial rows and/or columns of a matrix. Element referencing is performed via the '`[ ]`' operators, using the '`;`' to delimit row and column specifications, and the '`,`' to delimit individual row or column specifications.

To reference a single element:

```
> a = [1,2,3; 4,5,6; 7,8,9];
> a [ 2 ; 3 ]
        6
```

To reference an entire row, or column:

```
> a [ 2 ; ]
        4           5           6
> a [ ; 3 ]
        3
        6
        9
```

To reference a sub-matrix:

```
> a [ 2,3 ; 1,2 ]
        4           5
        7           8
```

As stated previously, any expression that evaluates to a matrix can have its elements referenced. A very common example is getting the row or column dimension of a matrix:

```
> size (a)[1]
        3
```

In the previous example the function `size` returns a two-element matrix, from which we extract the 1st element (the value of the row dimension). Note that we referenced the return value (a matrix) as if it were a vector. Referencing matrices in "vector-fashion" is allowed with all matrices. When vector-indexing is used, the matrix elements are referenced in column order. As with matrix indexing, a combination of vector elements can be referenced:

```
> a[3]
        7
> a[3,4,9]
        7           2           9
```

3.3.5 Assignment

Matrices can be assigned to in whole or in part. We have shown complete matrix assignment in the examples of the last few pages. In the same way that matrix elements can be referenced singly, or in groups, matrices can have single elements re-assigned, or groups of elements re-assigned. The result of an assignment expression is the left-hand-side (LHS). This is more convenient when working interactively, and when creating intermediate function arguments.

```
> a[2;2] = 200
 a =
        1           2           3
```

```
        4         200          6
        7           8          9
> a[2,3;2,3] = [200,300;300,400]
 a =
        1           2          3
        4         200        300
        7         300        400
```

The row and column dimensions of the matrices on the RHS, and the matrix description within the '`[ ]`' must have the same dimensions.

3.3.6 Matrix Operations

The usual mathematical operators `+,-,*,/` operate on matrices as well as scalars. For `A binop B`:

`+` Does element-by-element addition of two matrices. The row and column dimensions of both `A` and `B` must be the same. An exception to the aforementioned rule occurs when 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. An exception to the aforementioned rule occurs when either `A` or `B` is a 1-by-1 matrix; in this case a scalar-matrix addition operation is performed.

`*` Performs matrix multiplication on the two operands. The column dimension of `A` must match the row dimension of `B`. An exception to the aforementioned rule occurs when either `A` or `B` is a 1-by-1 matrix; in this case a scalar-matrix multiplication is performed.

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

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

The exception to the aforementioned dimension rule occurs when `A` 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 addition on its operands. The operands must have the same row and column dimensions. *Unless*:

- `A` or `B` is a 1x1. In this case the operation is performed element-by-element over the entire matrix. The result is a MxN matrix.
- `A` or `B` is a 1xN. and the other is MxN. In this instance the operation is performed element-by-element fashion for each row in the matrix. The result is a MxN matrix.

- `A` or `B` is a Nx1. and the other is NxM. In this instance the operation is performed element-by-element fashion for each column in the matrix. The result is a NxM matrix.

`.-` Performs element-by-element subtraction on its operands. The operands must have the same row and column dimensions. *Unless*:

- `A` or `B` is a 1x1. In this case the operation is performed element-by-element over the entire matrix. The result is a MxN matrix.
- `A` or `B` is a 1xN. and the other is MxN. In this instance the operation is performed element-by-element fashion for each row in the matrix. The result is a MxN matrix.
- `A` or `B` is a Nx1. and the other is NxM. In this instance the operation is performed element-by-element fashion for each column in the matrix. The result is a NxM matrix.

`.*` Performs element-by-element multiplication on its operands. The operands must have the same row and column dimensions. *Unless*:

- `A` or `B` is a 1x1. In this case the operation is performed element-by-element over the entire matrix. The result is a MxN matrix.
- `A` or `B` is a 1xN. and the other is MxN. In this instance the operation is performed element-by-element fashion for each row in the matrix. The result is a MxN matrix.
- `A` or `B` is a Nx1. and the other is NxM. In this instance the operation is performed element-by-element fashion for each column in the matrix. The result is a NxM matrix.

`./` Performs element-by-element division on its operands. The operands must have the same row and column dimensions. *Unless*:

- `A` or `B` is a 1x1. In this case the operation is performed element-by-element over the entire matrix. The result is a MxN matrix.
- `A` or `B` is a 1xN. and the other is MxN. In this instance the operation is performed element-by-element fashion for each row in the matrix. The result is a MxN matrix.
- `A` or `B` is a Nx1. and the other is NxM. In this instance the operation is performed element-by-element fashion for each column in the matrix. The result is a NxM 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*:

- `A` or `B` is a 1x1. In this case the operation is performed element-by-element over the entire matrix. The result is a MxN matrix.
- `A` or `B` is a 1xN. and the other is MxN. In this instance the operation is performed element-by-element fashion for each row in the matrix. The result is a MxN matrix.

- `A` or `B` is a Nx1. and the other is NxM. In this instance the operation is performed element-by-element fashion for each column in the matrix. The result is a NxM 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 via `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 matrix, 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.

Several details are important to note:

- The two character operators are just that, two characters. White space, or any other character in between the two symbols is an error, or may be interpreted differently.
- The expression `2./A` is **not** interpreted as `2. /A`. RLaB is smart enough to group the period with the '/'.

3.3.7 Matrix Relational Operations

The way matrices can be used within if-statement tests is special. The result of a matrix relational test, such as `A == B`, is a matrix the same size as `A` and `B` filled with ones and zeros according to the result of an element-by-element test. If either of the operands is scalar, or a 1-by-1 matrix, then the element-by-element test is performed as before, by using the scalar value repeatedly. For example.

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

RLaB if-tests do not accept matrices. The built-in functions `any()` and `all()` can be used in combination with relational and logical tests to conditionally execute statements based upon matrix properties. For example: perform a test that returns true or false (0 or 1) if `a` contains the value 4.

```
> any ( any (a == 4))
```

The function `any()` returns true if any of the element(s) of its argument are non-zero. The function `all()` returns true if all of the element(s) of its argument are non-zero. Note that `any` is used twice; this is because `any` is a vector-oriented function. This will be discussed later.

3.3.8 Examples

Now it is time for a few illustrative examples ...

Suppose you are learning about normal-equations, orthogonal transformations, QR decompositions, etc.[6] You have read the proper sections in your text(s), and you want to try your hand at it to see if you really understand it.

First you create an over-determined coefficient matrix, 3 parameters, and 5 equations (`a`). Then you create an observation matrix (`b`):

```
> a = [3,4,1;0,2,2;0,0,7;zeros(2,3)];
> b = [14;10;21;6;2];
```

You've just read that the RLaB operator '\' solves systems of equations, so you try it out:

```
> x = a \ b
 x =
        1
        2
        3
```

You check the answer (note that this is a contrived problem):

```
> b - a*x
-7.11e-15
-1.78e-15
-1.42e-14
        6
        2
```

and it looks "OK". The residual in the first three rows is near the machine-epsilon[7]. Now you wish to follow the example in the text more closely, in an attempt to reinforce your reading. The text has stated that the "normal equations" are: $(A^T A)x = (A^T b)$.

Not having read the chapter on Gaussian elimination, and matrix inverses yet you try:

[6] You've been reading Kahaner, Moler, and Nash "Numerical Methods and Software" for example.

[7] Machine-epsilon is the smallest number that your computer can distinguish. A common way to determine machine-epsilon is to divide a variable (`eps`) by two until `1.0 + eps == 1.0`.

```
> x = inv (a'*a) * (a'*b)
 x =
        1
        2
        3
```

Fortunately, the problem we are working with is not ill-conditioned, otherwise we may have produced a terrible result with the above procedure. If you want to pursue the reasoning behind the previous statement I suggest you read the section "Linear Systems of Equations".

Well, this is all too easy, now you want to get dirty, so you move on to orthogonal transformations. You have read about the construction of Householder vectors and reflections; now you would like to try it first-hand. You know that:

$$P = I - 2(vv^T)/(v^Tv)$$

Where v is the Householder vector used to form the reflection matrix. First you must construct the vector. Your text[8] tells you that a good method for constructing the Householder vector is:

$$v[2:n] = x[2:n]/(x[1] + sign(x[1]) * \|x\|_2)$$

$$v[1] = 1$$

```
> a = rand(5,2);                // Start out with a more difficult [A]
> a
 a =
    0.655      0.265
    0.129        0.7
     0.91       0.95
    0.112     0.0918
    0.299      0.902
> ac1 = a[;1];                  // grab the 1st column of [a] to work with
> u = norm (ac1, "2");          // compute the 2-norm of [ac1]
> v[2:5] = ac1[2:5] / (ac1[1] + sign (ac1[1])*u)
 v =
        0     0.0705      0.498     0.0611      0.164
> v[1] = 1;
> v = v';                       // make v a column vector
```

By using the matrix creation, and element referencing features you have generated the vector in 4 commands. We could have used a signal command

```
> v = [ 1 , a[;1][2:5]' / (a[1;1] + sign(a[1;1])*norm(a[;1],"2")) ]'
```

[8] Mine in this case is Golub and VanLoan, "Matrix Computations".

But, this is less than clear. Note that in this case, since we are working with vectors, we only use a single index when subscripting the variables.

Now that we have our Householder vector, we are ready to assemble the Householder reflection (matrix).

```
> P = eye (5,5) - 2*(v*v')/(v'*v)
 P =
   -0.558      -0.11     -0.776    -0.0952     -0.255
    -0.11      0.992    -0.0547   -0.00671     -0.018
   -0.776    -0.0547      0.614    -0.0474     -0.127
  -0.0952   -0.00671    -0.0474      0.994    -0.0156
   -0.255     -0.018     -0.127    -0.0156      0.958
> P*a
    -1.17       -1.2
-1.65e-17      0.596
-1.54e-16       0.22
-1.31e-17    0.00217
-5.39e-17      0.662
```

As you can see it worked out just like they said it would. All the elements of the first column of A, below the diagonal, have been zeroed out[9]. In this manner we can proceed to transform A into an upper triangular matrix.

[9] Although the numbers are not identically zero, they are on the order of machine precision, and can be considered to be zero. Some programs use fixed-point precision when printing. Although this makes their output look more accurate, it is not.

4 Program Flow Control

We must now take a small diversion before proceeding on with the rest of the objects and discuss flow-control. The flow-control statements available in RLaB are the *if-statement*, the *while-statement*, the *for-statement*, the *break-statement* and the *continue-statement*.

The flow-control statements use a syntax that is *similar* to the C-language[10]. Braces '`{`' and '`}`' are *required* in all of the flow-control statements.

4.1 If-Statement

The *if-statement* performs a test on the expression in parenthesis, and executes the statements enclosed within braces if the expression is true. The expression must evaluate to a scalar-expression. If the expression evaluates to a vector or matrix a run-time error will result.

```
if ( expression )
{
      statements
}
```

```
> if ( 1 ) { "TRUE" }
TRUE
> if ( 0 ) { "TRUE" }
```

An optional '`else`' keyword is allowed to delineate statements that will be executed if the expression tests false:

```
> if ( 0 ) { "TRUE" else "FALSE" }
FALSE
```

The `any` and `all` functions are useful with *if-statements*. If we want to execute some statements, conditional on the contents of a matrix:

```
> a=[1,2;3,0];
> if (!all (all (a))) { "a has a zero element" }
a has a zero element
```

4.2 While-Statement

The *while-statement* tests the expression in parenthesis, and executes the statements enclosed within braces until the expression is false. The expression must evaluate to a scalar-expression. If the expression evaluates to a vector or matrix a run-time error will result.

[10] Note we said similar, not identical.

while (*expression*)
{
    statements
}

```
> while ( 0 ) { "TRUE" }
> i = 0;
> while ( i < 2 ) { i = i + 1 }
 i =
        1
 i =
        2
```

4.3 For-Statement

The *for-statement* executes the statements enclosed in braces N times, where N is the number of values in *vector-expression*. Each time the statements are executed *variable* is set to the k^{th} value of *vector-expression*, where $k = 1 \ldots\ N$.

for (*variable* in *vector-expression*)
{
    statements
}

```
> for ( i in 1:3 ) { i }
 i =
        1
 i =
        2
 i =
        3
```

vector-expression can be any type of vector: real, complex, and string vectors are all acceptable.

4.4 Break and Continue Statements

The *break* and *continue* statements are simply keywords. Usage of *break* and *continue* is only allowed within *while-statements* or *for-statements*. *break* will cause execution of the current loop to terminate. *continue* will cause the next iteration of the current loop to begin.

```
> for ( i in 1:100 ) { if (i == 3) { break } } i
```

```
 i =
        3
> for ( i in 1:4 ) { if (i == 2) { continue } i }
 i =
        1
 i =
        3
 i =
        4
```

Although they will not be explicitly discussed - there are more examples of flow-control statement usage throughout the remainder of the primer.

5 Objects—Program Functions

Like matrices and strings, functions are stored as ordinary variables in the symbol table. 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) }
<user-function>
```

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.

```
> y = logsin
<user-function>
> y (2)
     0.63
>
> // Overwrite it with a matrix
> logsin = rand (2,2);
>
> // Check that y is still a function
> y (3)
    0.155
```

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 user-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
```

5.1 Function Syntax

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

> function (*argument list*)
> {
>
> *statements*
>
> }

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

There are several statements that only make sense within functions:

> global (*global-var-1, global-var-2* ...)
> local (*local-var-1, local-var-2* ...)
> return *expression*

The `global` and `local` statements are optional. There are no restrictions on the number of `local` or `global` statements or where they occur in the function. However, since these two statements only affect variables that are used *after* the declaration, it is recommended that you use `local` and `global` at the *beginning* of each function.

The `return` statement is also optional. There are no restrictions on the number of return statements, or their placement. A function can return from any point in its execution. The return statement must return a value. The value can be any RLaB object.

5.2 Function Scoping Rules

When you start a RLaB session, either interactively or in batch-mode, you create an environment. The environment or workspace consists of the built-in functions, and any other variables or functions you may have added. The workspace will also be referred to as the global-symbol-table or the global scope.

There are two other types of environment available: a function's local environment and a file's static environment[11]

A function's local scope is temporary, it is created when the function is invoked, and is destroyed when the function returns. A file's static scope is created when the file is loaded, and remains intact until the RLaB session is terminated.

The different scopes serve to protect data from operations that occur in the other scopes. There is some degree of overlap in order to allow flexibility. Functions can affect file-static and global scopes; statements within files can affect statements within other files and the global scope. More simply put, the "lower" scopes generally have access to the "higher" scopes. When a variable is used, RLaB uses certain rules to "bind" the variable. When we use the term bind or bound, we mean that the variable name is associated with an entry in one of the three types of symbol tables.

File-Scope: Variables that are in a file (but not within a function) are bound to the global-symbol-table (global-scope or global-environment) unless a static declaration is used. When a variable is declared static it is bound to the file's symbol table. From that point on, the variable will remain bound to the file's scope. When a variable is declared static, it is not visible from the global environment or from any other files.

Function Local Scope: In general, variables used within a function (other than the function's arguments) are bound to the function's local scope (there are ways to override this behavior). Variables bound to a function's local scope are not visible from a file's scope or from the global scope. They are created (undefined) when the function is invoked, and destroyed when the function returns.

There are exceptions: variables used in a function context are bound to the global-symbol-table. For example:

```
x = a * sin ( pi )
```

`sin` is used in a function context, and is bound to the global scope, while `x`, `a`, and `pi` are bound to the function's local environment.

Function's that are defined within a file have full access to the file's static variables. Function variables will be bound to the file's scope before local binding occurs.

```
---- beginning of file.r ----

static (A, pi)
```

[11] We will use the term environment and scope interchangeably.

```
pi = atan(1)*4;

fun = function ( a ) { return A*sin(pi*A*a); }

---- end of file.r ----
```

When 'fun' is created it binds 'A' and 'pi' to file.r's static environment.

There are two declarations: 'global' and 'local' that can be used to override the default behavior if necessary. Variables declared local will be bound to the function's local scope, and variable declared global will be bound to the global scope.

5.3 Function Argument Passing

Arguments can be passed by reference, or value. The default behavior is to pass by reference. Pass by references indicates that the variables used in the function call are directly referenced from within the function. Pass by value means that only the value of the argument is passed to the function. In other words, the arguments are copied. In order to pass a function argument by value, the user needs to declare that function argument as local. This method allows users to selectively determine how to pass each function argument.

A function can be called with fewer arguments than specified in the definition; this is called a "short-list". When this situation occurs, RLaB pads the argument list with extra undefined variables. Arguments can be "skipped" when calling a function. The "skipped" arguments are passed to the function as undefined variables. To "skip" an argument just leave it out of the argument list, but don't forget the commas:

```
> x = function ( a, b, c, d ) { a ? b ? c ? d ? }
<user-function>
> x (1); // short-list
        1
UNDEFINED
UNDEFINED
UNDEFINED
> x(1,,2); // skipped arguments
        1
UNDEFINED
        2
UNDEFINED
```

As far as RLaB is concerned undefined variables do not exist. The function `exist` will return true (1) if its argument exists, and false (0) if its argument is undefined.

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 classes and class-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 class and type, if necessary. The documentation (comments) should clearly define the requisite argument types and the function return object. 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 passed by value so that changes will not affect the caller's variables.

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 or built-in functions) as local. Declaring all function variables as local will prevent accidental destruction of user's global variables.

RLaB has a special built-in function (`fvscope`) that analyzes user-functions, and makes a report describing which variables are local, arguments, global, or file-static. `fvscope` can be very helpful when writing your first function(s) to help you understand how RLaB resolves variable references.

5.4 Function Recursion

Functions can call themselves recursively. Each time execution passes into the function the local variables are (re)created. There is a special keyword: `$self`, which must be used to force a function to refer to itself.

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

In the previous example a factorial computation is performed using recursion[12]. In the second return statement, the function calls itself until $a \leq 1$.

5.5 Files

Simple "one-liner" functions can be typed in at the command line. However, they are destroyed when the RLaB session is ended. Most users will want to create their functions in a text-editor as ordinary ASCII files.

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 its argument, it executes the RLaB statements in the file as if they were typed at the command line.

[12] Not necessarily an efficient way to compute the factorial.

Statements in a file are executed in the same manner as they would be had they been typed in interactively. Files containing ordinary commands and multiple functions are acceptable. 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
```

5.6 Example

Many functions are included with the RLaB source distribution. Functions can be found in the distribution subdirectories `./rlib`, `./toolbox`, and `./examples`. These directories are normally installed under `/usr/local/lib/rlab`.

We will continue with some simple examples demonstrating function creation and usage. We will carry on with the exercise of learning least-squares techniques.

In some earlier examples we played with solving a set of normal equations, and tried a simple experiment with Householder reflections. Now we want to try out this technique, and decompose a matrix into two matrices: Q and R; such that $A = QR$.[13]

We are going to decompose an entire matrix, so we will want to automate the procedures we used in previous examples. The first was creating a Householder vector. Instead of typing in our function at the command-line, we will use a text-editor to create the function in a file so that we can correct our mistakes without retyping the entire function.

```
//
// house_v(): Given an N-vector V, generate an n-vector V
// with V[1] = 1, such that (eye(n,n) - 2*(V*V')/(V'*V))*X
// is zero in all but the 1st component.
//

static (sign)

house_v = function(v)
{
  local(v)
```

[13] For an excellent discussion of the QR decomposition please refer to "Matrix Computations" Golub and Van Loan.

```
  n = max( size(v) );
  u = norm(v, "2");
  if(u != 0)
  {
    b = v[1] + sign(v[1])*u;
    if(n > 1)
    {
      v[2:n] = v[2:n]/b;
    }
  }
  v[1] = 1;
  return v;
};

sign = function ( X )
{
  if (X >= 0) { return 1; else return -1; }
}
```

Note that our new function is more complicated than our earlier "one-liner". This is due to the fact that the function is more efficient, and does some input-checking. Notice that the variables `b`, `n`, `u`, and `v` are *local*; these local variables will never be seen by the user, and will not interfere with any pre-existing variables by the same name in the global-workspace.

Also note that the function argument, `v` is copied to the function local variable `v`. This prevents the function from changing the values in the input argument, and thus destroying the contents of the caller's variable.

One other important feature of the new function is the usage of the `sign` function. `house_v` requires that the sign function return either 1 or -1. The RLaB built-in `sign` function will return zero when its argument is zero; this behavior is unacceptable, so we have written our own sign function. Declaring our new sign function to be static means that it will only affect statements within the file `house_v.r`.

To use our new function type:

```
> a = rand (4,2);
> x = house_v (a[;1])
 x =
        1
    0.434
    0.042
    0.412
```

Now that we can generate a Householder vector, we need to automatically form the Householder reflection, and use it to reduce A to upper triangular form.

```
// P.r
```

```
// P: Generate P matrix

P = function ( V )
{
  m = max( size(V) );
  return [ eye(m,m) - 2*(V*V')./(V'*V) ];
};
```

This function is a small one, and simply implements the formula we demonstrated earlier. We can test out our two new functions like so:

```
> ( p1 = P (house_v (a[;1])) ) * a
    -1.49      -1.11
-4.09e-17     0.0174
-2.87e-18      0.354
-8.89e-17     -0.779
> p1' * p1
        1   4.79e-17   2.75e-18    1.8e-17
 4.79e-17          1  -4.78e-18  -2.51e-18
 2.75e-18  -4.78e-18          1      9e-18
  1.8e-17  -2.51e-18      9e-18          1
```

Our new function seems to be working as expected. The computed Householder reflection, `p1`, zeros out all of the elements below the first in column one of `a`. Additionally `p1` is an orthogonal transformation, as demonstrated by computing ${P_1}^T P_1$. It is usually more efficient to build up programs as a collection of simple functions, like we are doing here, testing each as it is written, and making the appropriate fixes.

Function debugging can be easily accomplished by simply removing key ';' to turn expression printing on. Additionally, one can comment out `local` statements so that a function's variables can be examined after function execution.

Now there are two more pieces, a better implementation of `P` called `house_row`[14], and the final function (`house_qr`) that will apply the transformations in sequence to A to produce the upper-triangular R, and the orthogonal Q.

```
//
// house_row(): Given an MxN matrix A and a non-zero M-vector V
// with V[1] = 1, the following algorithm overwrites A with
// P*A, where P = eye(m,m) - 2*(V*V')/(V'*V)
//

house_row = function(A, v)
{
  local(A)
```

[14] See "Matrix Computations" by Golub and VanLoan.

```
  b = -2/(v'*v);
  w = b*A'*v;
  A = A + v*w';
  return A;
};

// house_qr.r
// Given A, with M >= N, the following function finds Householder
// matrices H1,...Hn, such that if Q = H1*...Hn, then Q'*A = R is
// upper triangular.

// House.qr returns a list containing `q', and `r'

rfile house_row
rfile house_v
rfile P

house_qr = function ( A )
{
  local (A)

  m = A.nr; n = A.nc;
  v = zeros(m,1);
  q = eye (m, m);

  for(j in 1:n)
  {
    // Generate the Householder vector
    v[j:m] = house_v( A[j:m;j] );

    // Apply the Householder reflector to A
    A[j:m;j:n] = house_row( A[j:m;j:n], v[j:m] );

    // Create Q
    if(j < m)
    {
      q = P ([ zeros (j-1,1); 1; v[j+1:m] ]) * q;
    }
  }
  return << q = q'; r = A >>;
};
```

Notice the three `rfile` statements near the top of the file. These statements ensure that the user-function dependencies are resolved *before* we try and execute the function. Also note how `house_qr` returns two matrices in a list, with element names `q`, and `r`. We can use, and test, this new function like:

```
> x = house_qr ( a )
   q            r
> x.q * x.r
      0.7       0.96
     0.95      0.915
   0.0918      0.441
    0.902     0.0735
> a
 a =
      0.7       0.96
     0.95      0.915
   0.0918      0.441
    0.902     0.0735
```

A visual comparison shows that our function does indeed work. Now we wish to use this factorization to compute a solution to our original least-squares problem. Since we have decomposed A into two matrices, one of which is upper triangular, we can reformulate the problem as a simple back-substitution. The RLaB built-in function `solve` will do this for us, since `sr` is already upper-triangular, all `solve` will do is the back-substitution.

```
> a = [3,4,1;0,2,2;0,0,7;zeros(2,3)];
> b = [14;10;21;6;2];
> x = house_qr (a);
> sq = x.q[;1:3];
> sr = x.r[1:3;];
> z = b'*sq;
> sol = solve (sr, z')
 sol =
        1
        2
        3
> b - a*sol
        0
        0
        0
        6
        2
```

Now that we have built our own `qr()` I should tell you that RLaB has a built-in `qr()` that is much more robust, and significantly faster.

6 Objects—Other Data Structures

Although RLaB is primarily a linear-algebra programming tool, other data structures are necessary to allow the user some flexibility, and a little extensibility. The two remaining data structures do just that. Strings allow users/programmers to write intelligible error messages, and properly annotate program inputs and outputs, as well as label quantities during program execution. The list object is a very flexible data structure that can hold numeric, string, function and other list objects. Since lists are indexed in an associative fashion, they are a very powerful tool.

6.1 Strings

When performing numeric computations, strings are not normally used for much except error messages and such. However, there are many other tasks for which strings are quite useful.

String objects can be treated in a manner similar to numeric objects, with the main difference being the different string operators. The only numeric operator that works on strings is the '`+`' which catenates the left and right string operands together.

```
> "first part of the string " + "and the second part of the string"
first part of the string and the second part of the string
```

The other numeric operators will only produce a runtime error message when used with strings.

```
> "string1" * "string2"
rlab: NULL, NULL, 1st arg invalid for multiply
```

The relational and logical operator do work with string objects. For instance, you can compare two strings:

```
> "string-a" == "string-b"
        0
```

Or two string matrices:

```
> ["el-1", "el-2"] == ["el-1", "el-x"]
        1          0
```

Functions exist to aid with string to number conversions (`num2str`), and reading strings from the standard input (`input`).

```
> x = input ("input a string > ", "s")
input a string > teststring
teststring
```

The function **strsplt** will split an arbitrary string into a matrix of one-character strings. This allows the user to operate on individual characters in a string; the '+' operator can be used to "glue" the desired pieces back together again.

```
> x = "this is a single string";
> strsplt (x)
t  h  i  s     i  s     a     s  i  n  g  l  e     s  t  r  i  n  g
```

We will do a little more with strings in Section 6.2.

6.2 Lists

A list is a heterogeneous associative array. Simply, a list is an array whose elements can be from different classes. Thus a list can contain numeric, string, function, and other list objects. Lists have many uses, only some of which we will demonstrate herein.

To create a list-object use the '<<' and '>>'' operators. The list will be created, and the objects inside the '<< >>' will be copied into the new list. If the objects are not renamed during the list-creation, they will be given numerical index values.

```
> a = rand(3,4); b = sqrt (a); c = 2*a + b;
> ll = << a ; b ; c >>
  1            2            3
> ll2 = << A = a; b = b ; x = c >>
  A            b            x
> ll2.A == ll.[1]
        1          1          1          1
        1          1          1          1
        1          1          1          1
```

Lists are not indexed in what is perceived as the "traditional manner". Instead the list index is converted to a string, and the string value is used to look-up the referenced object[15]. There are two methods for referencing the elements of a list, the first, a shorthand notation looks like:

list_name . element_name

In this case, the *list_name* and *element_name* must follow the same rules as ordinary variable names. The second method for indexing a list is:

list_name . [numeric_or_string_expression]

[15] RlaB lists are similar to AWK associative arrays.

The second method allows string and numeric variables to be evaluated before doing the conversion to string type[16].

As you have seen in an earlier example lists can be used within functions when it is necessary to return several values. Because of the lists flexible nature a function can return matrices of differing sizes and types, as well as strings, other lists, or user-functions.

6.2.1 Examples

The `getline` function reads from a designated file, or pipe, and separates the input into whitespace separated tokens. The tokens are either numbers or strings. In this example we will use the `getline` function, and RLaB piping capability to read the output from the UNIX ps command, and sum the numbers in the 5th column (the resident process size).

```
> psize = 0;
> while (length (ans = getline ("|/usr/ucb/ps -aux | grep ian")))
  {
    psize = psize + ans.[5];
  }
> close ("|/usr/ucb/ps -aux | grep ian");
> psize
 psize =
 2.26e+03
```

The following simple example makes extensive use of the list object's capabilities. This example consists of two user-functions that were written to allow someone to index a matrix with strings, or numeric quantities that are not necessarily equal to the integer row and column numbers.

Two special qualities of lists are used in this example. The first is the ability to group together related objects; thus the matrix, and its row and column labels are copied into a list-object. The second special quality is the associative nature of the list indices. When constructing the row and column labels, we create two lists that are members of the top-level list. These sub-lists contain the row or column label as the index value, and the integer row or column number as data.

```
//
// Create a list-object, a matrix with row and
// column labels. Then the matrix can be indexed
// with the labels by using lb().
//

mklb = function (mat, rl, cl)
{
  lrl = <<>>; lcl = <<>>;
  if (!exist (rl))
```

[16] If one is necessary.

```
  {
    rl = 1:mat.nr;
  else
    if (length (rl) != mat.nr) { error ("mklb: rl wrong size"); }
  }

  for (i in 1:rl.n)
  {
    lrl.[rl[i]] = i;
  }

  if (!exist (cl))
  {
    lcl = 1:mat.nc;
  else
    if (length (cl) != mat.nc) { error ("mklb: cl wrong size"); }
  }

  for (i in 1:cl.n)
  {
    lcl.[cl[i]] = i;
  }

  return << m = mat; rl = lrl; cl = lcl >>
};
```

the function `mklb` creates a list containing matrix, row, and column labels. Looking closely at the for-loops you will notice that the labels values are actually the index values of the lists `lrl` and `lcl`. This method is used since it provides extremely easy access to the matrix elements in the next function `lb`.

```
lb = function ( mlb, rl, cl )
{
  // Create row indices

  if (!exist (rl))
  {
    irl = 1:mlb.m.nr; // use all the rows
  else
    for (i in 1:rl.n)
    {
      irl[i] = mlb.rl.[rl[i]];
    }
  }

  // Create column indices
```

```
  if (!exist (cl))
  {
    icl = 1:mlb.m.nc; // use all the columns
  else
    for (i in 1:cl.n)
    {
      icl[i] = mlb.cl.[cl[i]];
    }
  }

  return mlb.m[irl; icl]
};
```

The for-loops in `lb` simply loop through the specified indices, using them as index values to the internal row and column lists. Once the numeric vectors `irl` and `icl` have been formed the specified matrix elements are easily returned.

To use the new functions you might:

```
> m = [1,2,3;4,5,6;7,8,9];
> ml = mklb (m, ["r1","r2","r3"], ["c1","c2","c3"])
   cl           m            rl
> lb (ml, ["r1","r3"])
        1          2          3
        7          8          9
> lb (ml, ["r1","r3"], "c1")
        1
        7
> lb (ml, , "c2")
        2
        5
        8
```

7 Builtin functions

RLaB comes with many built-in and user-functions. The built-in functions are available any time RLaB is run, regardless of the command-line options. The user-functions that are delivered with RLaB may, or may not be accessible, depending upon how RLaB has been configured on your computer and the command-line options used.

Typing '`rlab -ql`' will run RLaB without executing the `.rlab` file or loading any of the delivered user-functions. The built-in functions in RLaB are unlike user-functions in that users cannot destroy or reassign them to other variables[17]. This protection is in place so that users will be able to write

[17]The exception to this statement is the use of static variables, which can obscure built-in functions - see the Section 5.6.

portable libraries using the built-in functions.

RLaB does *not* automatically search `RLAB_SEARCH_PATH` for R-files when an unresolved reference is encountered. Instead, the user must explicitly `load`, or use the `rfile` command to get RLaB to load and compile the function or statements. To see what functions are currently available type: '`what()`', this will list all of the currently available functions. Typing '`rfile`' from within RLaB will list all of the R-files in the directories listed in the environment variable `RLAB_SEARCH_PATH`. The files displayed by the `rfile` command can be loaded by typing '`rfile  filename`', where `filename` is the name of the R-file to load, *without* the `.r` extension.

7.1 Function Behavior

Many of the delivered functions fall into one of three categories: scalar-functions, vector-functions, or matrix-functions. The list below gives examples from each type of function, but is not all-inclusive.

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 $m \geq 1$ then the operation is performed in a column-wise fashion. Some examples are:

sum	cumsum	prod	cumprod
mean	max	min	fft
sort	any	all	ifft

When using a vector oriented function, like `max()` it is possible to operate on matrix objects. For example the maximum value in a matrix can be obtained via `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
factor	backsub	issymm	

Misc. Functions: These functions are in this particular category simply because they don't fit anywhere else. Some examples are:

system	getline	show	who
what	tic	toc	printf
format	write	read	readm

8 Input and Output

There are two types if I/O in RLaB .

1. Raw data, such as matrices, lists, and strings. In this case the formatting is irrelevant.
2. Formatted I/O, such as data from other programs, and formatted output intended for reports and such.

In either case file names, or file-handles, are the same. A file-handle is *always* a string, either a string constant, a string variable, or a string expression. For example:

```
> write ("file.rd", A);

> f = "file.rd"; write (f, A);

> write ("file" + ".rd", A);
```

are all equivalent.

There are three special file-handles in RLaB ; they are: `"stdin"`, `"stdout"`, and `"stderr"`.

`"stdin"` is connected to RLaB 's standard input, usually the keyboard. `"stdout"` is connected to RLaB 's standard output, usually the screen. And `"stderr"` is connected to the standard error, again, usually the screen.

Files are automatically opened by the functions that perform input or output. In most cases the files will automatically be closed after the function has completed its task. The following will force a file closure after their task is complete: `rfile`, `load`, `read`, and `readm`.

However, `write`, `writem`, `fprintf`, and `getline` will leave their files open after completion, so that they may be used subsequently without complicated file positioning operations.

RLaB has another special type of file-handle. When a string, starting with a '`|`' is used as a file-handle, a process is created. Either the sub-process input, or output is connected to RLaB through the file-handle. For example:

```
> output = "| sort";
```

```
> for (i in 5:1:-1) { fprintf (output, "%i\n", i); } close (output);
1
2
3
4
5
```

If the fprintf output were not sent to the Unix program sort, you would expect the output to appear in descending order. In fact, fprintf does write the output in descending order, to sort. Then, sort re-orders its input, and writes it to stdout. The same trick works in reverse; functions that expect input can get it from a sub-process. A good example is getline:

```
> input = "| ls -la *.[ch]";
> fsum = 0;
> while (length (ans = getline (input))) { fsum = fsum + ans.[5]; } fsum
 fsum =
   943083
> close (input);
```

In this instance we used the sub-process capability to determine the number of bytes of C source code, and header files exist in the current working directory. The sub-process performs the long listing, and getline parses this input into numbers and strings. The 5^{th} column of the ls -la output is the number of bytes in each file.

9 Plotting

There are two available methods for plotting data from within RLaB . Probably the most desirable is the usage of the builtin plotting library, Plplot. The second method involves the usage of RLaB 's I/O abilities (piping) to send data and commands to another running process - Gnuplot, for example. A fairly extensive example of this second method is provided in the RLaB source distribution in the file misc/gnu_plot.r. We shall focus on the builtin plotting capability.

There are two new terms you must learn to understand plotting - plot-window, and sub-plot. The plot window is manifested on the plot-device, such as a Tektronix screen, X-window display, or a printer. RLaB will allow you to have multiple plot-windows, if your display device will support it. X-windows supports multiple plot-windows, but a Tektronix display, or DOS display will not.

The next term is sub-plot. Each plot-window can have 1 or more sub-plots. The layout of the sub-plots is determined when pstart is called.

Regardless of the number of plot-windows and sub-plots, there can only be one *current* plot-window, and one *current* sub-plot within the current plot-window. Each plot related function will only work on the current sub-plot in the current plot-window. This way the majority of the plot functions are the same, regardless of the output device. The current sub-plot is always the *next* plot. As soon as a plot function is used that actually draws the plot on the plot-device, such as plot, plot3 or plhist, the current sub-plot is incremented. Therefore, all functions that effect the appearance of a plot must be used before the plot is drawn.

9.1 2-D Plotting

To plot some data with a minimum of effort do:

```
> data = (0:pi:.1)';
> data[;2] = cos (2*pi*data*3);
> pstart (,,"xwin");
> plot (data);
```

The `pstart` function was called with the default sub-plot definition (1 subplot), and the X-windows driver as the display device. The `plot` function accepts matrices, or lists as arguments and plots the columns of a matrix, or the columns of each matrix of a list.

Now for a slightly more complex example.

```
> t = (0:10:.05)';
> x = exp (-0.5*t) .* (cos (2*pi*3*t) + sin (2*pi*7*t));
> X = fft (x);
> rfile faxis
> freq = faxis (X, .05, 3);
> mag = abs (X);
> rfile angle
> phase = angle (X);
> pstart (1,2,"xwin");
>
> ptitle ("Magnitude of FFT")
> xlabel ("Frequency (Hertz)");
> plot ([freq,mag]);
>
> ptitle ("Angle (atan2(imag/real)) of FFT")
> xlabel ("Frequency (Hertz)");
> ylabel ("Angle (radians)");
> plot ([freq,phase]);
> plprint ("p3.ps");
```

In this example we create a plot-window with two subplots so that we can display related information on the same plot-window. The arguments to `pstart` specify that there will be 1 plot in the horizontal direction, and 2 plots in the vertical direction, thus creating 2 sub-plots. The first `ptitle` and `xlabel` function calls effect the first sub-plot. The first call to `plot` creates the first sub-plot. The subsequent calls to `ptitle`, `xlabel` and `ylabel` effect the second sub-plot, and the last call to `plot` creates the second sub-plot.

The last line is a call to `plprint`. The `plprint` function creates a file that contains a copy of the contents of the current plot-window. The default hardcopy device for `plprint` is Postscript,

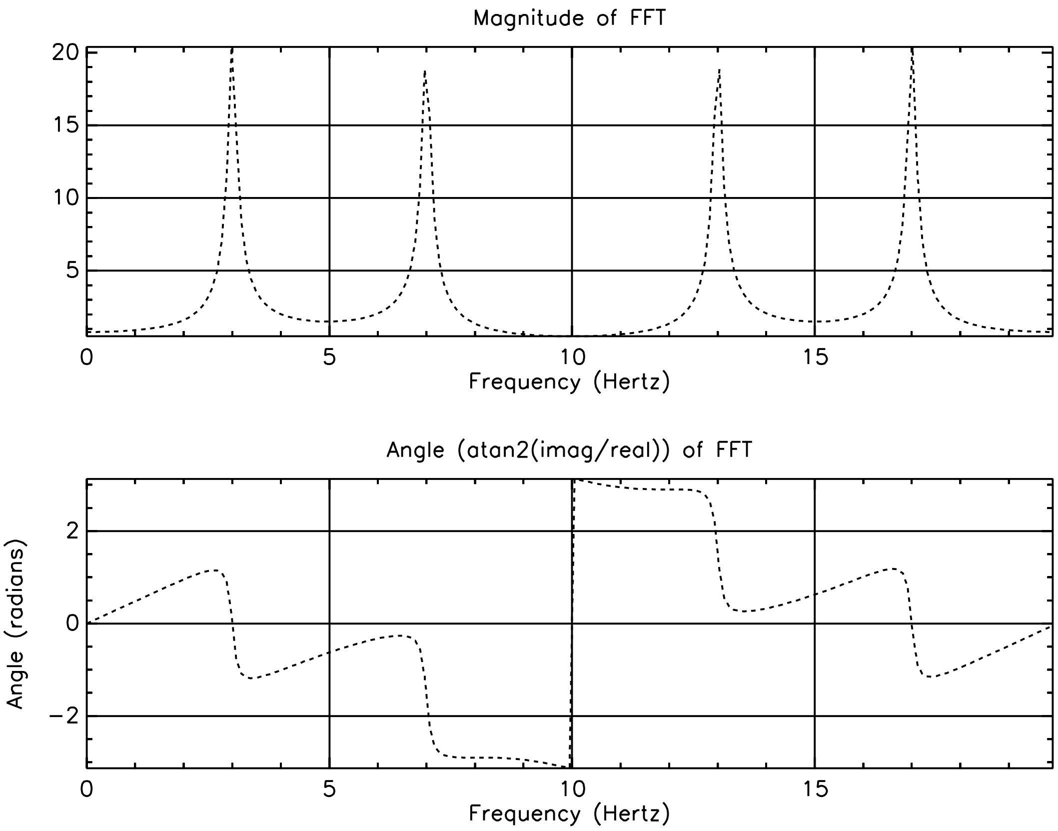


Figure 2: Example Plot

but color Postscipt, Xfig, plmeta, and HP-LJII can also be specified. The output from `plprint` is presented in Figure 2.

9.2 Histograms

Histograms can be plotted quite easily. At present the histogram plotting function, `plhist`, will plot histograms of the columns of a matrix. For example:

```
> // Create data
> rand("normal", 0, pi);
> r = rand(2000,1);
>
> // Create plot
> ptitle("Histogram, 30 Bins");
> plhist (r, 30);
```

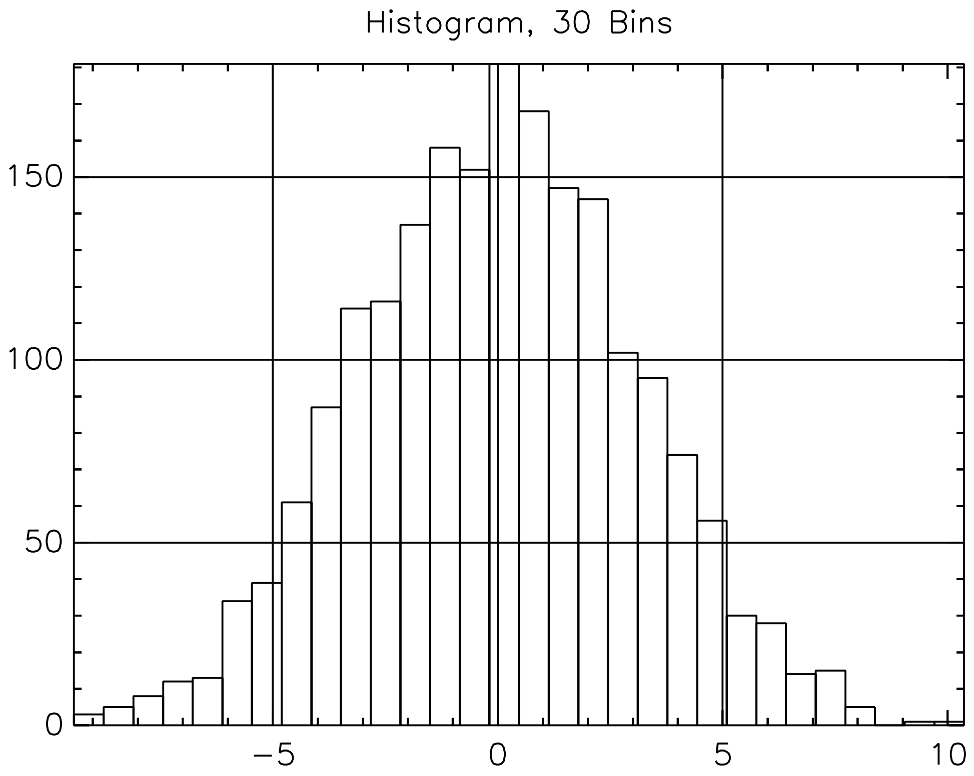


Figure 3: Example Histogram Plot

Figure 3 contains the Postscript output from the previous example.

9.3 3-D Plotting

Plotting 3-dimensional data is quite straightforward, and very similar to the steps used for plotting 2-dimensional data. We will begin with an example:

```
> // Create the data
> X = -2:2:.2;
> Y = X;
> Z = [];
> for (i in 1:X.n) {
>   for (j in 1:Y.n) {
>     Z[i;j] = X[i] * exp (-X[i]^2 - Y[j]^2);
>   }
> }
> // Make the plot
> pstart(,,"xwin");
> plwid(4);
> ptitle("Sample 3-D Plot");
> xlabel("X-Axis");
> ylabel("Y-Axis");
> zlabel("Z-Axis");
> plmesh( << x = X; y = Y; z = Z >> );
> plprint("3d.ps");
```

The output from the plprint command appears in Figure 4. Now we must take a minute and explain the data passed as input to `plot3`. Three dimensional plots have two independent variables (vectors), `x`, and `y`. The dependent variable `z` (a matrix) is a function of both `x` and `y` and has row dimension the length of `x` and column dimension the length `y`. This storage scheme for three-dimensional data is economical in terms of memory, and allows the user some extra flexibility. Thus, the argument to `plot3` is a list with three elements: `x`, `y`, and `z`. `plot3` can accept up to three distinct lists for plotting.

The same plot-functions: `ptitle`, `xlabel`, and `ylabel` can be used with 3-dimensional plots. Additionally, `zlabel` can be used to add labels to the `z` axis.

The next example (Figure 5) demonstrates more plotting capabilities - including: changing fonts and pen widths, multiple plots per page and annotating a plot. Note that the calls to `plptex` are made *after* the `plot` calls. since `plptex` uses the plot coordinates to place text on the graph, the plot must be created first.

```
//
// Demonstrate the effect of adding terms to a Fourier expansion
//

// We want to approximate a square wave...
// The Fourier Series for a square wave is a
// sum of odd harmonics - we will demonstrate.
```

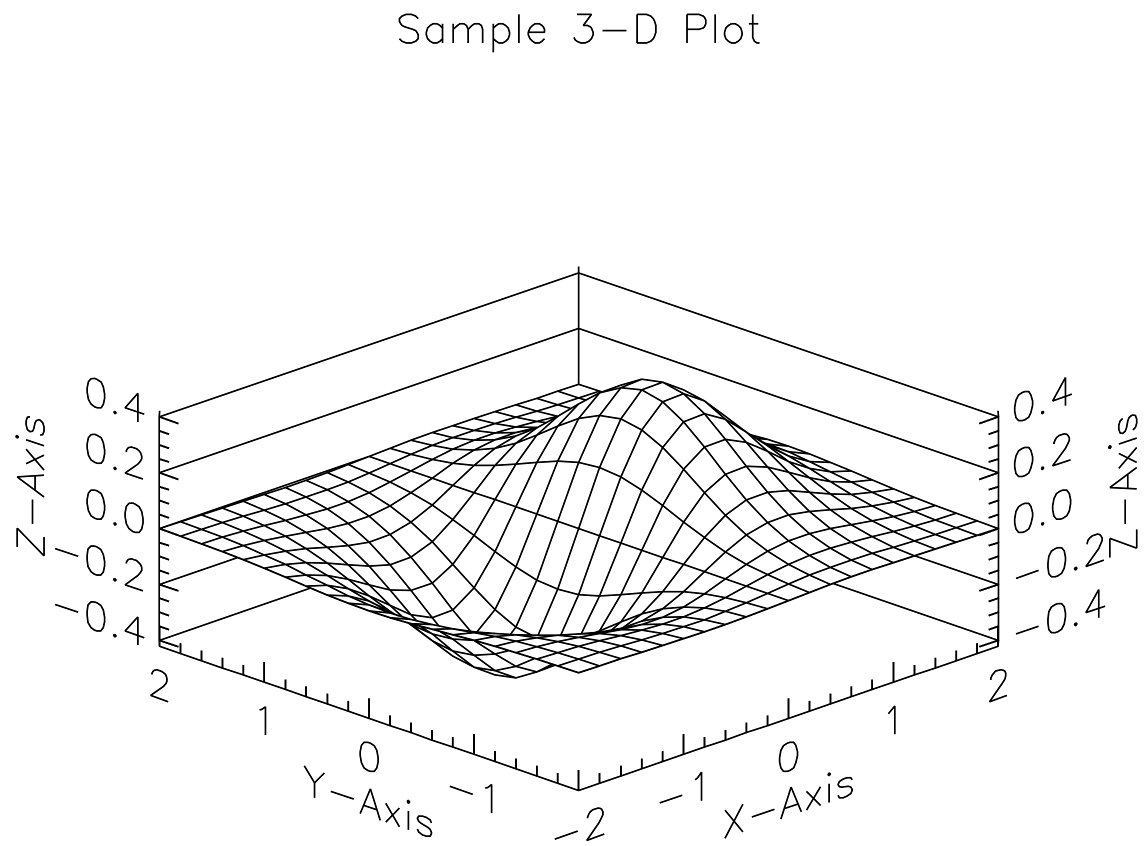


Figure 4: Example 3D Plot

```
// Start up a plot window...

pstart(2,2, "xwin");

// Compute the 1st term in the Fourier series and plot.

t = (0:10:.1)';
y = sin (t);

ptitle ("Fundamental Frequency");
plwid(10);
plfont(1);
plot ( [t,y] );
plptex ("Pen width = 10", 4, 0.4);
plptex ("Font = 1 (Normal)", 4, 0.1);
pause ();

// Now add the third harmonic to the fundamental, and plot.

y = sin (t) + sin (3*t)/3;

ptitle ("1st and 3rd Harmonics");
plwid(7);
plfont(4);
plot ( [t,y] );
plptex ("Pen width = 7", 4, 0.4);
plptex ("Font = 4 (Script)", 4, 0.1);
pause ();

// Now use the first, third, fifth, seventh, and ninth harmonics.

y = sin (t) + sin (3*t)/3 + sin (5*t)/5 + sin (7*t)/7 + sin (9*t)/9;
ptitle ("1st, 3rd, 5th, 7th, 9th Harmonics");
plwid(4);
plfont(3);
plot ( [t,y] );
plptex ("Pen width = 3", 4, 0.4);
plptex ("Font = 3 (Italic)", 4, 0.1);
pause ();

//
// Now create a matrix with rows that represent adding
// more and more terms to the series.
//

t = (0:3.14:.02);
```

```
y = zeros(10,max(size(t)));
x = zeros(size(t));

for (k in 1:19:2)
{
  x = x + sin(k*t)/k;
  y[(k+1)/2;] = x;
}

//
// Now make a nice 3-D plot that shows the effect
// of adding more and more terms to the series.
//

plfont (2);
plwid(1);
ptitle ("Square Wave via Fourier Series");
ylabel ("No. Terms");
plaz (140);
plot3 (<< x = t; y=1:10; z=y' >>);
plptex ("Pen width = 1", -1, 4.5);
plptex ("Font = 2 (Roman)", -1, 3.8);
plprint ("p6.ps"); // Make hardcopy (Postscript default).
```

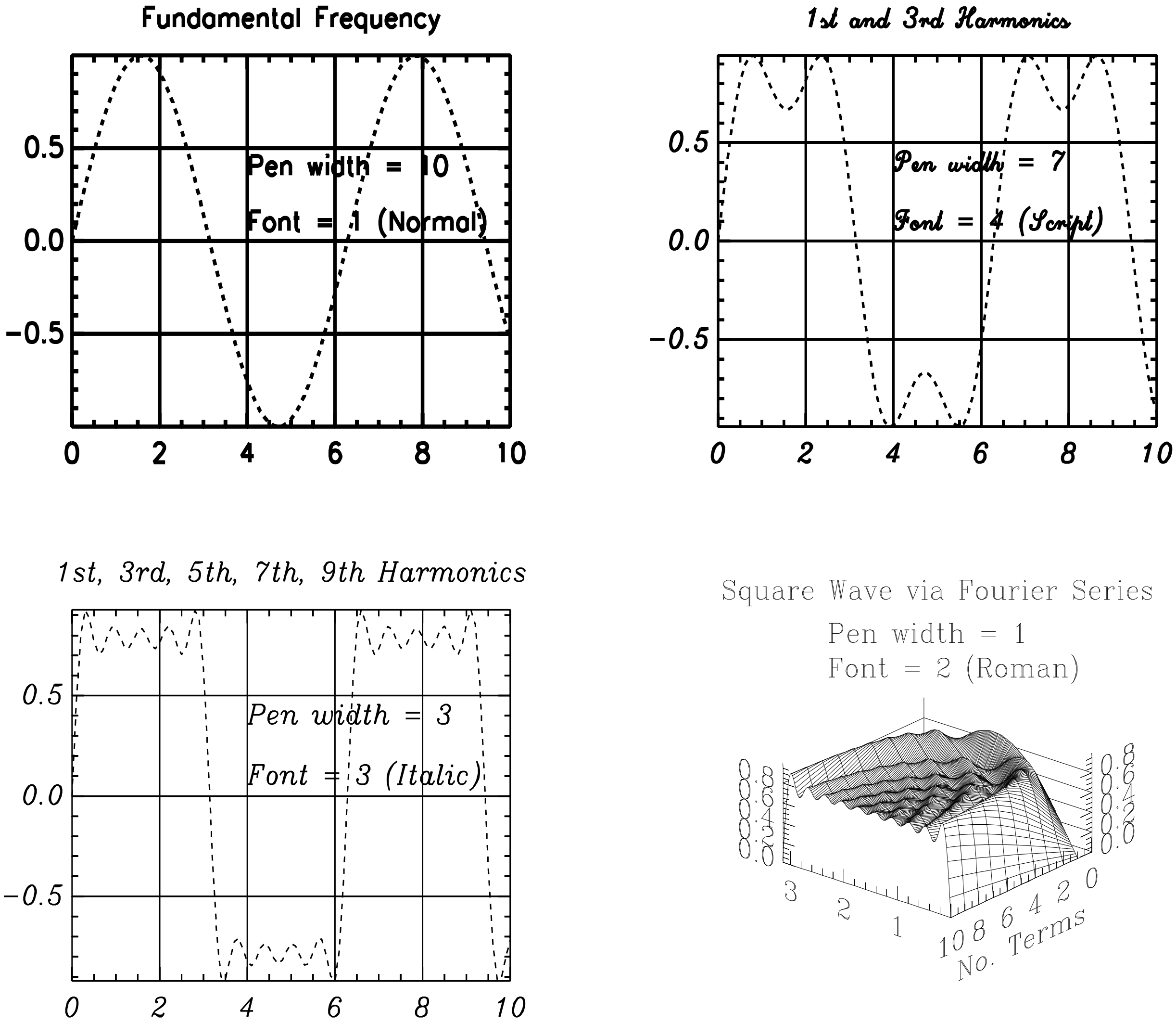


Figure 5: Square Wave Plot Example

10 Summary

RLaB can do quite a few things that we have not covered here. The reference manual or the online help should be consulted for more detailed descriptions of language and function behavior. We will close out this primer with a listing of the currently available function (both built-in and user-functions).

```
> what ()
abs          epsilon      log          plot         solve
acos         error        log10        plot3        sort
acosh        eval         logspace     plprint      sprintf
all          exist        lu           plptex       sqrt
any          exp          lyap         plstyle      srand
asin         eye          matrix       plwid        std
asinh        factor       max          printf       strsplt
atan         fft          maxi         printmat     strtod
atan2        filter       mean         prod         sum
atanh        find         members      pstart       svd
backsub      finite       min          ptitle       sylv
balance      fix          mini         pwin         symm
cd           floor        mod          qr           system
ceil         format       nan          rand         tan
chol         fprintf      norm         rank         tanh
class        fvscope      num2str      rcond        tic
clear        getb         ode          read         tmp_file
clearall     getenv       ones         readb        toc
close        getline      open         readm        trace
compan       hess         pause        real         tril
complement   hilb         pclose       redit        triu
conj         ifft         pend         replot       type
cos          imag         plalt        reshape      union
cosh         inf          plaspect     round        what
cross        input        plaxis       save         who
cumprod      int          plaz         scalar       whos
cumsum       int2str      plegend      schord       write
det          intersection plfont       schur        writeb
diag         inv          plgrid       set          writem
diary        isempty      plgrid3      show         xlabel
diff         isinf        plhist       showpwin     ylabel
disp         isnan        plhistx      sign         zeros
dot          issymm       plhold       sin          zlabel
eig          length       plhold_off   sinh
eign         linspace     plimits      size
eigs         load         plmesh       sizeof
> quit
```