Ian R. Searle & Phillip Musumeci

Contents

0 Primer priming

0.1 RL.3ex aB is freely available

RL.3ex aB 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.

0.2 Acknowledgments

The availability of kfreel software, such as GNU Emacs, GNU gcc and gdb, gnuplot, and the Netlib archives has made this project possible. The RL.3ex aB author thanks both the authors and sponsors of the LAPACK, RANLIB, and FFTPACK projects.

0.3 Document reproduction and errors

The RL.3ex aB Primer is freely available. Permission is granted to reproduce the document in any way providing that it is distributed for free, except for any reasonable charges for printing, distribution, staff time, etc. Direct commercial exploitation is not permitted. Extracts may be made from this document providing an acknowledgment of the original LATEX source is maintained.

We welcome reports of errors and suggestions for improvement in this document and also in RL.3ex aB . 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

RL.3ex aB is written in C. The maths libraries used are written in Fortran but the use of a publicly available FortranC converter reduces compiler requirements to C (the conversion tool f2c is written in C). The core of the data display system, gnuplot, is publicly available in C source code form (and even binaries for some PCs). This makes the whole RL.3ex aB 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

RL.3ex aB 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. RL.3ex aB 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.

RL.3ex aB is a **structured language**

These days, computer languages do look very similar but we will try to point out a few *useful* similarities! which will be familiar to users of C and also the Wirth-inspired languages such as Pascal and Modula. An RL.3ex aB 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 ksubprograml in RL.3ex aB 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 permanently available to all of your subprogram functions and it is also possible to declare data to be local to a function - such local function storage exists only for the duration of the call of the function, 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 krun awayl 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 RL.3ex aB .

RL.3ex aB features strongly typed objects but with the emphasis on usefulness, not on pedantics. In RL.3ex aB 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 RL.3ex aB 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 RL.3ex aB 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 RL.3ex aB session staring up at you. After showing you how to run RL.3ex aB and get on-line help, we describe data types before moving back to a khands onl description of basic operations. Program structure is then described and you will see how to write your own functions. Finally, we look at flow control and looping. As RL.3ex aB comes 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 RL.3ex aB to develop your own programs.

2 Starting to use RL.3ex aB

2.1 How to run it

A properly installed RL.3ex aB 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.

RL.3ex aB will start with a message similar to:

```
Welcome to RLaB. New users type `help INTRO'
RLaB version 0.NN beta, 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 RL.3ex aB command prompt. At this point, userjs 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 RL.3ex aB session you can type `quit` at the RL.3ex aB prompt. On Unix systems an `EOF` or `^d` (control-d) will also stop RL.3ex aB .

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 RL.3ex aB . 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 RL.3ex aB script files which we refer to as kR-filesl - those marked *rlib* come as a standard part of RL.3ex aB and the remainder refer to local R-files that have been setup for you by whoever installed your RL.3ex aB .

In general, the functions listed in the first group are the most efficient as they are compiled into the core of RL.3ex aB . In contrast, RL.3ex aB js 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 RL.3ex aB . If an R-file feature is really useful, it can be added to the core RL.3ex aB program since you have the source code.

2.3 Simple calculations

RL.3ex aB is designed for mathematical calculations so letjs 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 [illegible] system, as well as the interests of the person installing RL.3ex aB . such as when * (`inf`) is a result or an input to further calculation; and also knot-a-numberl (`NaN`). RL.3ex aB can use complex numbers as well as real numbers so now try

```
> 1/1i
   0 - 1i
> 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 . 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 RL.3ex aB , variables can have names of any length containing most printable characters including . You will observe that we have to exclude special characters such as `+, -, *, /` and the `SPACE` character. The actual assignment operator symbol is k=l and an assignment statement looks like

```
 variable_name = expression_to_evaluate
```

and an example is

```
> radius=2
 radius =
   2
> circumference=2 * pi * radius^2
 circumference =
25.1
```

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 RL.3ex aB 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 i;j character. Termination an expression with a i;j will suppress printing of the result. Likewise, terminating an expression with the i?j 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. RL.3ex aB 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 ?.

[illegible], then you might still be able to take advantage of command line recall and editing. Try typing the * key to see if any previous RL.3ex aB commands are displayed - if they are, then confirm that * also displays more recent commands and then try horizontal cursor movement with the 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 werenjt 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 RL.3ex aB command line, you will find that are the same as `C-p C-b 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 RL.3ex aB may have been built without command line editing - this is unlucky.

3 Objects - Basic Data Structures

In the most general form, an object in RL.3ex aB can be data or a function - a fact that no doubt excites the hormones in the modern day object oriented programmer. It is even possible to construct an object that contains both data *and* functions. We are going to discuss basic data types before looking at how data can be kgroupedl together for some useful purpose. We will also work through some simple examples that manipulate data but first, what does RL.3ex aB regard as data?

3.1 Data Types

There are three *fundamental* types of data that you manipulate in RL.3ex aB : the string; the real number; and the complex number. As we have seen in section ?, 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 kkeep it simplel, RL.3ex aB which is primarily concerned with [illegible], enclose the characters inside quotes like `"this"` e.g.

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

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 RL.3ex aB , *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 RL.3ex aB and it is now possible to understand a little more about how data structures and functions are organised within RL.3ex aB .

3.2 Object Hierarchy

Scan your eyes down over Figure ? which shows the hierarchical structure of objects in RL.3ex aB - we shall now describe this figure from the bottom up (ignoring lists until a little later). Not all objects are created the same 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 RL.3ex aB instructions can manipulate.

Figure 1: RL.3ex aB objects

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 RL.3ex aB what its class is e.g. RL.3ex aB has a built-in command to calculate the sin of an angular quantity - asking RL.3ex aB 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 RL.3ex aB - expect gratuitous use of these functions as further examples are given. We are particularly interested in exploring the use of RL.3ex aB as a computation tool so now we describe further numeric operations.

3.3 Numerics

The RL.3ex aB numeric object 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 =
   123
   456
   789
```

In this context the i`[ ]`j signal RL.3ex aB that a matrix should be created. The inputs (or arguments) for matrix creation are whatever is inside the i`[ ]`j. The rows of the matrix are delimited with i`;`j and the elements of each row are delimited with i`,`j.

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 i`[ ]`j.

```
> 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()` 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 `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 RL.3ex aB , 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 i`:`j operator(s), that is i`start:end:inc`j. The leftmost operand, `start`, specifies the starting value, the second operand, `end`, specifies the last value. The default increment, or spacing, is 1. But, a third operand, `inc`, can be used to specify a non-unity increment.

```
> v = 1:4
 v =
   1234
```

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)
   35
> 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 kread-onlyl. 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;
```

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

```
> a = reshape (a, 1, 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 i`[ ]`j operators, using the i`;`j to delimit row and column specifications, and the i`,`j 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 ; ]
   456
> a [ ; 3 ]
   3
   6
   9
```

To reference a sub-matrix:

```
> a [ 2,3 ; 1,2 ]
   45
   78
```

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 kvector-fashionl 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]
    729
```

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 =
    123
    4   2006
    789
> a[2,3;2,3] = [200,300;300,400]
 a =
    123
    4   200   300
    7   300   400
```

The row and column dimensions of the matrices on the RHS, and the matrix description within the i[ ]j 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 kright-divisionl 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 kleft-divisionl with the arguments transposed.

  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, RL.3ex aB has several other operators that function on matrix operand(s).

- `.*` Performs element-by-element multiplication on its 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 its operands. The operands must have the same row and column dimensions, unless either `A` or `B` is a 1-by-1 matrix.
- ***** Performs matrix kleft-divisionl. 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 via an Ajs 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`. RL.3ex aB is smart enough to group the period with the i/j.

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
    100
    010
    001
> a >= 5
    000
    011
    111
```

RL.3ex aB 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 matrix equations, orthogonal transformations, QR decomposition, etc. You have been reading Kahaner, Moler, and Nash Numerical Methods and Software for example. 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];
```

Youjve just read that the RL.3ex aB operator i\j 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 machine-epsilon. Machine-epsilon is the smallest number the computer can distinguish. A common way to determine machine-epsilon is to divide a variable (`eps`) by two until `1.0 + eps  == 1.0`.. 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 knormal equationsl are: .

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

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

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:

Where P is the Householder reflection matrix. Mine in this case is Golub and Van Loan, Matrix Computations. First you must construct the vector. Your text tells you that a good method for constructing the Householder vector is:

$$v[1]=1$$

```
> 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
    0.1120.0918
    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 =
   00.0705 0.4980.0611 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. 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 the numbers just below the diagonal are effectively zero. All the elements of the first column of A, below the diagonal, have been zeroed. Although the numbers are not identically zero, they are on the order of machine precision, and can be considered zero. Some programs use fixed-point precision when printing. Although this makes their output look more accurate, it is not.. In this manner we can proceed to transform A into an upper triangular matrix.

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 RL.3ex aB 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. Braces i{j and i}j are *required* in all of the multiple-expression 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 i`else`j 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.

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 value of *vector-expression*, where *k*=1? *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. And, like other variables in the symbol table, functions are accessible as global variables. Functionjs 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)
```