Debuging and Linear Systems

Lecture 9

- Moving Scripts to function
- Debuging
- Matrix Operation
- Solving Linear Systems

Programming and Debugging

Functions: Function forms the core of R; everything you do in R uses a function in one way or another. More importantly, the way functions work in R allows you to do complex operations in one step or a few simple steps. You can call built-in function or write your own function. A good programmer is always to use functions for their codes.

Function: Name Arguments

We can also call the function using named arguments. When calling a function in this way, the order of the actual arguments doesn't matter. For example, all of the function calls given below are equivalent.

```
pow(8, 2)
```
[1] "8 raised to the power 2 is 64"

```
pow(x = 8, y = 2)
```
[1] "8 raised to the power 2 is 64"

```
pow(y = 2, x = 8)
```
[1] "8 raised to the power 2 is 64"

Default Values for Arguments

We can assign default values to arguments in a function in R.
This is done by providing an appropriate value to the formal argument in the function declaration.

Here is the above function with a default value for y.

```
pow <- function(x, y = 2) {
  # function to print x raised to the power y
    result <- x^y
    print(paste(x,"raised to the power", y, "is", result))
}
```

The use of default value to an argument makes it optional when calling the function.

```
pow(3)
```
[1] "3 raised to the power 2 is 9"

```
pow(3,1)
```
[1] "3 raised to the power 1 is 3"

Return Value from Function

We will require our functions to do some processing and return back the result. This is accomplished with the return() function in R.

- Syntax of return()
- return(expression)

The value returned from a function can be any valid object. Let us look at an example which will return whether a given number is positive, negative or zero.

```
check <- function(x) {
   if (x > 0) {result <- "Positive"  }
   else if (x < 0) { result <- "Negative"}
   else { result <- "Zero" }
   return(result)
}
```

Here, are some sample runs.

```
check(1)
[1] "Positive"
 check(-10)
[1] "Negative"
 check(0)
[1] "Zero"
```

Functions Without return()

If there are no explicit returns from a function, the value of the last evaluated expression is returned automatically in R.

For example, the following is equivalent to the above function.

```
check <- function(x) {
   if (x > 0) {
        result <- "Positive"
   }
 else if (x < 0) {
        result <- "Negative"
   }
   else {
        result <- "Zero"
   }
   result
}
```

Multiple Returns

The return() function can return only a single object. If we want to return multiple values in R, we can use a list (or other objects) and return it.

```
 circle<-function(r){
   area<-pi*r^2;
   circumference<-2*pi*r;
   mylist<-list(a1=area, a2=circumference)
   return(mylist)
}
  a<-circle(3)
  area<-a$a1;
  circumference<-a$a2
```

Or

```
 circle<-function(r){
   area<-pi*r^2;
   circumference<-2*pi*r;
   mylist<-c (area, circumference)
   return(mylist)
}
 circle(3)
```

Programming and Debugging

Readability: It is important to keep your code readable. You wrote the code yourself, so you should know what it does

You do now, but will you remember what you did if you have to redo that analysis six months from now on new data. Besides, you may have to share your codes with other people. To keep your code readable

Programming and Debugging

To keep your code readable

- Add comments which are not read by programming.
- Make objects name clear and descriptive. Names should contain only letters, numbers, underscore characters(_) and dot(.)
- Cannot use the following special keywords as names: -- *False, for, function*, *if, Inf, Na, NaN, next, NULL, repeat, return, TRUE, while*
- Keep the flow of your code. Call functions at appropriate place. You can also include funtions within you main code.

Debugging: Three Kind of Bugs

- Syntax errors: if you write code that R cannot understand, you have syntax errors. Syntax errors always result in an error message and often are caused by misspelling a function or forgetting a bracket.
- Semantic errors: If you write correct code that does not do what you think it does, you have a semantic error. The code itself is correct but the outcome of the code is not.
- Logic errors: probably the hardest-to-find errors. Your code works, it does not generate any errors or warning, but it still does not return the result you expect. The mistake is not code itself, but in the logic it executes.

Debugging: Three Kind of Bugs

- If something goes wrong with your code, R tells you in two ways:
  - The code keeps on running until the end, and when the code is finished, R print out a warning message.
  - The code stops immediately because R cannot carry it out, and R prints out an error message.

Debugging

Reading error messages

Let’s take a look at such an error message. If you try the following code, you get a more or less clear error message:

```
"a" + 1
Error in "a" + 1, non-numeric argument to
binary operator
```

You get two bits of information in this error message. First, the line "a" +1, tells you in which line of code you have an error. Then it tells you what the error is. In this case, you used a non-numeric argument (the character "a"): In combination with a binary operator (the +sign).

Debugging

Error messages aren't always that clear. Take a look at the following example:

```
data.frame(1:10, 10:1,)
Error in data.frame(1:10, 10:1, ) :
argument is missing, with no default
```

After the second vector, there's a comma that shouldn't be there. A small typing error, but R expects another argument after that comma and doesn't find one.

If you don't immediately understand an error message, take a closer look at the things the error message is talking about. It could be that you simply typed something wrong there.

Warnings

Warnings often are the only sign you have that your code has some semantic or logic error. Here is a warning that pops up regularly and may point to a semantic or logic error in your code:

```
x <- 4
sqrt(x - 5)
[1] NaN  Warning message: In sqrt(x - 5) : NaNs produced
```

Because x-5 is negative when x is 4, R cannot calculate the square root and warns you that the square root of a negative number is not a number (NaN).

A simple syntax error can generate a warning instead of an error. For example,

```
plot (1:10, 10:1, color = "green")
Warning messages: 1: In plot.window(. . .) : "color" is not a
graphical parameter 2: In plot.xy(xy, type, . . .) : "color" is
not a graphical parameter ....
```

If you get warning or error messages, a thorough look at the Help pages of the function(s) that generated the error can help in determining what the reason is for the message you got. For example, at the Help page ?plot.xy, you find that the correct name for the argument is col.

Going Bug Hunting

A grammatically correct program may give you incorrect results due to logic errors. In case such errors (i.e. bugs) occur, you need to find out why and where they occur so that you can fix them. The procedure to identify and fix bugs is called “debugging”. R provides a number of tools for debugging such as:

traceback()

debug()

browser()

Going Bug Hunting: `traceback()`

Bug hunting a complex business, but some simple strategies can help you track down these pesky creatures. One simple tool that is sometimes useful in R is the traceback() function. If a program fails, and it's not clear where or why, invoking traceback() may provide useful information. Take an example

```
pow <- function(x, y) {
result <- x^y
print(paste(x,"raised to the power", y, "is", result))
check()
result
}
 check <- function() {
 my_list <- lit("color" = "red", "size" = 20, "shape" = "round")
 return(my_list)
}
```

If we call function pow, there is an error since it calls check where there is an error (list is misspelled lit).

Going Bug Hunting: `traceback()`

```
[1] "3 raised to the power 3 is 27"
Error in lit(color = "red", size = 20, shape =
"round") :
      could not find function "lit
```

If you only examines the function “pow”, you may be confused by the error message since there is no lit in function pow. However one uses traceback() to trace where is the error from.

```
traceback()
2: check() at pow.R#5
1: pow(3, 3)
```

The above information clearly shows that the error occurs in check(), line 5.

Going Bug Hunting: `Debug()`

traceback() does not tell you where in the function the error occurred. In order to know which line causes the error, you may want to step through the function using debug().

- flags the function for debugging debug(function name).
- unflags the function undebug(function name) .
- when a function is flagged for debugging, each statement in the function is executed one at a time.
- after a statement is executed, the function suspends and you can interact with the environment.
- After you obtained necessary information from the environment, you can let the function to execute the next statement.
- In this way, you can check the function line-by-line.

Going Bug Hunting: Debug ( )

```
## compute sum of squares
SS<-function(mu,x) {
d<-x-mu
d2<-d^2
ss<-sum(d2)
Ss
}
# set seed to get reproducible results
set.seed(100)
x<-rnorm(100)
SS(1,x)
[1] 202.5615
```

```
## now start debugging
debug(SS)
SS(1,x)
debugging in: SS(1, x)
debug: {
d <- x - mu
d2 <- d^2
ss <- sum(d2)
ss
}
```

After you see the "Browse[1]>" prompt, you can do different things:

- Typing n executes the current line and prints the next one;
- By typing Q, we can quit the debugging;
- Typing "where" tells where you are in the function call stack;
- By typing ls(), we can list all objects in the local environment;

Going Bug Hunting: `browser()`

If you place a call to browser() inside a function, execution will pause when the call is reached, allowing you to examine and modify variables. All valid R statements can be used. It is similar to debug() except you can control where execution gets paused. Suppose we have the following function:

```
doit = function(x,y){
 z = x + y
browser()
return(sum(z > 10)) }
```

Now suppose we invoke the function:

```
doit(1:8,5:12)
Called from: doit(1:8, 5:12)
 Browse[1]> objects()
[1] "x" "y" "z" Browse
[1]  x
[1] 1 2 3 4 5 6 7 8
```

Matrices

- A matrix is a rectangular array of numbers (or functions).

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & \\ \vdots & & \ddots & \\ a_{m1} & & & a_{mn} \end{bmatrix}$$

- The matrix shown above is of size $m \times n$. Note that this designates first the number of rows, then the number of columns.
- The elements of a matrix, here represented by the letter 'a' with subscripts, can consist of numbers, variables, or functions of variables.

Vectors

- A vector is simply a matrix with either one row or one column. A matrix with one row is called a row vector, and a matrix with one column is called a column vector.
- Transpose: A row vector can be changed into a column vector and vice-versa by taking the *transpose* of that vector. e.g.:

$$if \quad A = \begin{bmatrix} 3 & 4 & 5 \end{bmatrix} \quad then \quad A^T = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}$$

Matrix Addition

- Matrix addition is only possible between two matrices which have the same size.
- The operation is done simply by adding the corresponding elements. e.g.:

$$\begin{bmatrix} 1 & 3 \\ 4 & 7 \end{bmatrix} + \begin{bmatrix} 6 & 2 \\ 3 & 1 \end{bmatrix} = \begin{bmatrix} 7 & 5 \\ 7 & 8 \end{bmatrix}$$

Matrix scalar multiplication

- Multiplication of a matrix or a vector by a scalar is also straightforward:

$$5 * \begin{bmatrix} 1 & 3 \\ 4 & 7 \end{bmatrix} = \begin{bmatrix} 5 & 15 \\ 20 & 35 \end{bmatrix}$$

Transpose of a matrix

- Taking the transpose of a matrix is similar to that of a vector:

$$\textit{if} \quad A = \begin{bmatrix} 1 & 3 & 8 \\ 4 & 7 & 2 \\ 6 & 5 & 0 \end{bmatrix}, \quad \textit{then} \quad A^T = \begin{bmatrix} 1 & 4 & 6 \\ 3 & 7 & 5 \\ 8 & 2 & 0 \end{bmatrix}$$

- The diagonal elements in the matrix are unaffected, but the other elements are switched. A matrix which is the same as its own transpose is called *symmetric*, and one which is the negative of its own transpose is called *skew-symmetric*.

Matrix Multiplication

- The multiplication of a matrix into another matrix not possible for all matrices, and the operation is ***not commutative***:

$$AB \neq BA \text{ in general}$$

- In order to multiply two matrices, the first matrix must have the same number of columns as the second matrix has rows.
- So, if one wants to solve for C=AB, then the matrix A must have as many columns as the matrix B has rows.
- The resulting matrix C will have the same number of rows as did A and the same number of columns as did B.

Matrix Multiplication

- The operation is done as follows:

  using index notation: $$C_{jk} = \sum_{l=1}^{n} A_{jl} B_{lk}$$

- for example:

$$AB = \begin{bmatrix} 4 & 3 \\ 7 & 2 \\ 9 & 0 \end{bmatrix} \begin{bmatrix} 2 & 5 \\ 1 & 6 \end{bmatrix} = \begin{bmatrix} 4*2+3*1 & 4*5+3*6 \\ 7*2+2*1 & 7*5+2*6 \\ 9*2+0*1 & 9*5+0*6 \end{bmatrix}$$

$$= \begin{bmatrix} 11 & 38 \\ 16 & 47 \\ 18 & 45 \end{bmatrix}$$

Linear Systems of Equations

- One of the most important application of matrices is for solving linear systems of equations which appear in many different problems including statistics, and numerical methods for differential equations.
- A linear system of equations can be written:

$$a_{11}x_1 + \ldots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + \ldots + a_{2n}x_n = b_2$$
$$\vdots$$
$$a_{m1}x_1 + \ldots + a_{mn}x_n = b_m$$

- This is a system of m equations and n unknowns.

Linear Systems of Equations

The system of equations shown on the previous slide can be written more compactly as a matrix equation:

$$Ax=b$$

where the matrix A contains all the coefficients of the unknown variables from the LHS, x is the vector of unknowns, and b a vector containing the numbers from the RHS

Transform these Equations into Matrix

- $2x + 3y - z = 6$
  $-x - y - z = 9$
  $x + y + 6z = 0$

- $x + y = 0$
  $y + z = 3$
  $z - x = 2$

Create variables in R for each matrix

Gauss Elimination

- Example:

  $-x + 2y = 4$

  $3x + 4y = 38$

- first, divide the second equation by -2, then add to the first equation to eliminate y; the resulting system is:

  $-x + 2y = 4$

  $-2.5x = -15 \rightarrow \quad x = 6$

  $\rightarrow \quad y = 5$

Linear Equations in R

Create Matrix i.e. A

```
A <- as.matrix(data.frame(c(1,-1,0),c(1,2,1),c(-2,1,-1)))
```

Element-wise Matrix Multiplication

```
A*A
```

Identity Matrix (3x3)

```
diag(3)
```

Calculate inverse of matrix

```
solve(A)
```

Matrix Determinant

```
det(A)
```

Transpose of Matrix

```
t(A)
```

Find eigen values

```
e <- eigen(A)
e$values # eigne values
e$vector
```