Introduction to computational programming
Chapter 1 Lab Exercise
Functions and Algorithms

David M. Rosenberg
University of Chicago
Committee on Neurobiology

Version control information:
Last changed date: 2009-10-05 15:42:55 -0500 (Mon, 05 Oct 2009)
Last changes revision:80
Version: Revision 80
Last changed by: David M. Rosenberg

October 12, 2009

Overview

Part I

Tutorial

1 Functions

One of the most fundamental of programming concepts is that of the *function*, a series of commands which take several values (variables) as input and return the result of computations on that input. It is generally helpful (and frequently necessary) to place a series of related commands in a function. *R* lets you define functions using the following syntax

```
> functionName <- function(argumentList) {
+   codeBlock
+   {return(returnValue)}
+ }
```

Breaking the components down, we have

1. *functionName* is the name you would like to assign the new function to.[1].

[1] See Appendix 1 for more on function name style

2. *argumentList* the inputs to the function
3. *codeBlock* the commands to be evaluated
4. *returnValue* (optional) the output value that is "returned" by the function

As an example, let us define a function which takes as input an integer, `k` and returns the first number in the fibonacci sequence equal to or greater than that number.

```
> fibLargerThan <- function(k) {
+   x0 <- 0
+   x1 <- 1
+   while(x1 < k) {
+     newX1 <- x0 + x1
+     x0 <- x1
+     x1 <- newX1
+   }
+   return(x1);
+ }
> fibLargerThan(100)

[1] 144

> fibLargerThan(1000)

[1] 1597

> fibLargerThan(10000)

[1] 10946
```

1.1 Anatomy of a function

name This is the "command" that the function is bound to and is found to the left of the assignment operator in the function definition.

```
fibLargerThan <- function(x) { ...
```

arguments These are the variables passed (copied) into the function. Only those variables given as arguments to the function may be used within it. More about argument and scoping will be discussed below.

```
fibLargerThan <- function(x) { ...
```

body The body is the where the "work" of the function takes place. It is a free standing *block* of *R* code (enclosed by braces).

```
fibLargerThan <- function(k) {
  x0 <- 0
  x1 <- 1
  while(x1 < k) {
    newX1 <- x0 + x1
    x0 <- x1
    x1 <- newX1
  }
  return(x1);
}
```

return The return value is the "output" of the function. There can only be a single "thing" returned from any function call[2].

```
return(x1);
```

1.2 Scope

As stated above, only[3] those variables passed to a function as arguments and variables created in the *body* of the function are accessible by the function. Furthermore, variables created inside a function "disappear" once the function returns its result. This dichotomy between variables "accessible" to a function and those accessible outside constitute the notion of variable *scoping* or *scope*.

Many beginning programmers find variable scope to be a confusing and frustrating topic. It is suggested (until you become more comfortable with variable scope[4]) that you try not to use the same variable names both "inside" and "outside" of a function.

1.3 Functors

Functors are the name given to functions which take *other functions* as an argument. The following example demonstrates the use of functors.

In this example, the builtin function `curve()` is a *functor* – it takes the function `parabfun()` as its first argument. There are two ways you can build your own functors in *R* .

`apply` **family** The `apply` family of functions (named `apply`, `sapply`, `mapply`, ...) take a function name (or expression) as their *second* argument and *apply* that function to each member of their first argument (which should be a vector or list). The `sapply()` function operates on vectors, `lapply()` on lists, and `mapply()`. See the online help documentation and examples for details.

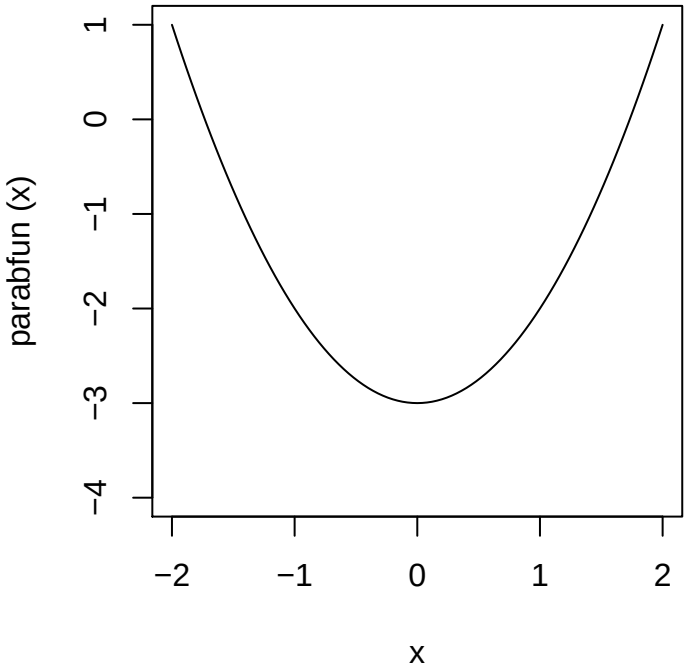
`do.call()` The `do.call()` function takes to arguments: a function (or expression) is its first argument and a named list of parameters as its second. It then evaluates the given function with the listed parameters.

[2]If you need to return more than one value, use a `list`, `vector`, or `data.frame` to encapsulate them.

[3]To be fair, this is a bit of a generalization. But its good practice.

[4]These simplified explanations aren't completely correct. *R* uses a system called *lexical scoping* over *functional closures* to define which variables are "in" scope and which are not. These rules are confusing and (at times) non-intuitive. They are beyond the scope of this course.

```
> ## Expand on previous plot ...
> parabfun <- function (x) {
+   return(x^2-3)
+ }
> curve(parabfun, xlim=c(-2, 2),
+   ylim=c(-4, 1))
```




```
> myFun <- function (x) {
+   return(x^2 - 1)
+ }
> myOtherFun <- function (x, y, z) {
+   return(min(c(x/y, x/z, y/z, z/y, z/x, y/x)))
+ }
> x_vector <- 1:10
> x <- 1
> y <- 2
> z <- 3
> sapply(x_vector, myFun)

 [1]  0  3  8 15 24 35 48 63 80 99

> do.call(myOtherFun, list(x=x, y=y, z=z))

[1] 0.3333333
```

2 Pseudocode translation

The process of transforming a pseudocode outline into a functional implementation starts with a basic understand of algorithm itself. Once you are confident that you understand what the algorithm *means*[5], you are ready to begin. Below is a *generalized* method transforming pseudocode into an implementation.

1. Try to identify and separate distinct "parts" of the algorithm and the relationships between them. I often find it helpful to look specifically for
   (a) Groups of operations which can naturally form a *function*
   (b) Inputs and outputs
   (c) Loops
   (d) Branches[6].
2. Identify where (and what type of) variables will be needed. I often find it helpful to write down the names I will use for my variables write next to the pseudocode.
3. Identify the *termination conditions* - the ways in which the algorithm will stop. For iterative algorithms, this is often a maximum number of loops or evidence of convergence.
4. Start writing and remember, there's no need to go in order! Feel free to write whatever "part" seems most natural to you.[7].

To help you get started, let's work work through the first pseudocode algorithm from the text. Hopefully, "seeing" how the process works may make it easier. We will consider here the specific scenario described in the first example, where we derived the difference equation.

$$N_{t+1} - N_t = 4N_t$$

Furthermore, let us consider an initial population size of 10 ($N_0 = 10$).

3 Example: numerical solutions for difference equations

An algorithm for numerically solving a difference equation is given in the text. It is printed again below for reference.

Pseudocode for numerical solution of difference equations

1. Define the iterated map function `F(x)`
2. Choose an initial condition x_0 and store it as the first element of the array x (we will use $x[0]$, but in some languages it has to be $x[1]$)

[5] A good test of whether you *understand* an algorithm is whether you could (in theory, with enough time) perform the algorithm with paper and a pencil. Computers are not magic.

[6] Branching describes the situation wherein a the results of a *conditional* (`if` statement) pick between two substantially different options

[7] Programming is a lot like doing a 1000 piece puzzle. Sometimes you are very uncertain about the first couple pieces you put down and. Often you will move or change them later. But the more pieces you have "on the board", the easier it is to see how the rest fit in.

3. For i starting at 1, repeat until i exceeds the specified number of iterations
   (a) Assign $x[i]$ the value of `F(x[i-1])`
   (b) increment i by 1

More compactly, we can write this as

```
define function F(x)    // iterated map function
real x0                 // initial condition
int N                   // total iterations
real array X [1..N]
X[0] ← x0               // assign first value
for i in 1 to N do
begin
  X[i] ← F(x[i-1])
end
```

Pseudocode listing 1 – Numerical difference equation algorithm

On first pass, I break the algorithm down into its "component" parts and note the names for variables I intend to use.

`define function F(x)  // iterated map function`	`lets call this function diffEqImap`

`real x0  // initial condition`
`int N  // total iterations`
`real array X [1..N]`
`X[0] ← x0  // assign first value`
	`call it init_value (numeric)`
	`max_iter (numeric)`
	`x_vector (numeric)`

`for i in 1 to N do`
`begin`
`  X[i] ← F(x[i-1])`
`end`
	`use ii as our counter`
	
	`termaination condition`

Here I have broken the algorithm down into three separate "chunks" (represented by the boxes) and notated the variable names I will use.

Writing the first "chunk" (the iterated map function) should be no problem, since we already know it from the text!

```
> diffEqImap <- function(x_at_t) {
+   x_at_t1 <- 5 * x_at_t;
+   return(x_at_t1);
+ }
```

No problems there. Continuing, the next "chunk" is merely a bunch of variable assignments.

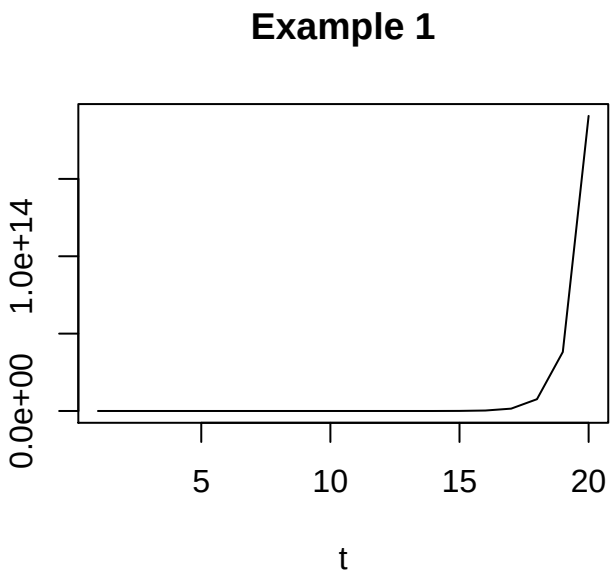
```
> init_value <- 10;
> max_iter <-20;
> x_array <- numeric(length=max_iter);  # size of x_array determined by max_iter
> x_array[1] <- init_value;             # R indexing begins at 1, not 0
```

Nothing too bad there, either. Note how I allocated the array `x_array` to its full size all at once, rather than starting with a vector of length one and "growing" it later. When possible, this is preferred.

```
> for (ii in 2:max_iter) {
+   new_value <- diffEqImap(x_array[ii-1]);
+   x_array[ii] <- new_value
+ }
```

At this point we're all done. Since the pseudocode algorithm didn't specify and output, the plot below shows the resulting values for `x_array`.

```
> ## Expand on previous plot ...
> plot(1:20, x_array, type='l',
+      main='Example 1', xlab='t',
+      ylab='x(t)');
```



Part II

Exercises

1. Write a function that takes as input a polynomial[8] and returns the polynomial representing the first derivative of the input polynomial. You will be provided two functions, `parsePolynomial()` and `deparsePolynomial` to help you in this task. The function `parsePolynomial()` takes as input a polynomial in x and returns a vector of exponents and a vector of coefficients. The function `deparsePolynomial()` takes as input a vector of exponents and a vector of coefficients and returns a polynomial in x.

[8] Helper functions are included at the end of the exercise.

2. *Logistic Model.*

   (a) Consider the logistic model $f(x) = x\left(2 - \frac{x}{2}\right)$. Calculate and plot the numerical solution and fixed points of this difference equation..

   (b) Write a *functor* which takes as an argument a function which is a logistic difference equation. Your functor should find the fixed points of the difference equation and plot its solution as an iterated map.

3. Using the functor from the previous exercise, plot iterated logistic of the following models and analytically find the fixed points of the models. State whether the solution found computationally approaches the analytically calculated fixed points.

   (a) $f(x) = 2x(1 - x)$

   (b) $f(x) = 4x(1 - x)$

```
#!/usr/bin/env rr
# encoding: utf-8
# parsePolynomial.R
#
# parsePolynomial - a function for parsing a polynomial (in x) into a
#             vector of coefficients and a vector of exponents.
# argument - inputExpression - a polynomial in x, expressed as a string.
#             For example,   "x^3 + 2x^2 - x - 1"
# returns  - a list with two components, coefficients and exponents,
#             representing the coefficients and exponents of the polynomial,
#             respectively.
#
# Example:
#       > inputEx <- "x^3 + 2x^2 - x - 1"
#       > result <- parsePolynomial(inputEx)
#       > result[['exponents']]
#       [1] 3 2 1 0
#       > result[['coefficients']]
#       [1] 1 2 -1 -1
#
parsePolynomial <- function(inputExpression) {
  numberTerms <- nchar(gsub("[^x]", "", inputExpression));
  coefficients <- numeric(length=(numberTerms)+1);
  exponents <- numeric((length=numberTerms)+1);
  splitTerms <- strsplit(gsub('-', '+ -', inputExpression), '\\+')[[1]]
  for (ii in seq(along=splitTerms)) {
    term <- gsub(' ', '', splitTerms[ii]);
    if(gsub("[^x]", "", term) == '') {
      term <- paste(term, 'x^0', sep='');
    }
    if (gsub("([\\+|-]*)(.*)x.*", "\\2", term) == '') {
      term <- gsub('x', '1x', term);
    }
    if (gsub(".*x", "", term) == '') {
      term <- paste(term, '^1', sep='');
    }
    coef <- eval(as.integer(gsub('x.*', '', term)));
    coefficients[ii] <- coef;
    exponent <- eval(as.integer(gsub('.*\\^(.*)', '\\1', term)));
    exponents[ii] <- exponent;
```

```
  }
  o <- -(order(exponents)) + length(exponents) + 1;
  exponents <- exponents[o];
  coefficients <- coefficients[o];
  result <- list(exponents=exponents, coefficients=coefficients);
  return(result);
}
# deparsePolynomial - a function which takes a vector of coefficients and a
#             vector of exponents and constructs a polynomial.
# arguments - coefficients - a vector of integer coefficients
#             exponents - a vector of exponents
#
# Example:
#       > exponents <- c(3, 2, 1, 0)
#       > coefficients <- c(1, 2, -1, -1)
#       > deparsePolynomial(coefficients, exponents)
#       [1] "x^3 + 2x^2 - x - 1"
#
deparsePolynomial <- function(coefficients, exponents) {
  if (length(exponents) != length(coefficients)) {
    stop('Exponent vector and coefficient vector must be of equal length.\n')
  }
  out_string <- '';
  for (ii in seq(along=exponents)) {
    out_string <- paste(out_string, ' + ', as.character(coefficients[ii]),
                        'x^', as.character(exponents[ii]), sep='');
  }
  out_string <- gsub('\\+ -', '- ', out_string);
  out_string <- gsub('1x', 'x', out_string);
  out_string <- gsub(' x\\^0', ' 1', out_string);
  out_string <- gsub('x\\^0', '', out_string);
  out_string <- gsub('^ [\\+|-] ', '', out_string);
  out_string <- gsub('x\\^1', 'x', out_string);
  return(out_string);
}
```