Part A Simulation and Statistical Programming HT15

Lecturer: Geoff Nicholls

University of Oxford

Lecture 6: functions and flow control (in R)

Notes and Problem sheets are available at

www.stats.ox.ac.uk\ ∼nicholls\PartASSP

Overview for lecture 6

1. R functions, function arguments and scoping

2. Flow control with `for()`, `if()-else` and `while()`.

3. Examples (Sieve of Eratosthenes and Newton's method).

Functions: what and why

What: functions take input, perform operations on the input and return output. Functions help us break down a big problem into modules. We can get each function working accurately and build up a working system. Functions help us avoid repeating the same sequence of commands in lots of different places.

The general definition of a function is :

```
MyFunction = function(arguments) expression
```

The arguments can defined with defaults. When the function is called the arguments are passed to the expression for evaluation.

A simple `expression` is a `statement` (like `a<-pi^2/2`). A compound expression is a sequence of statements in braces: {`statements`}. A function expression returns the last evaluated statement. We will use the `return()` function to end expressions to spell out to the reader what the function returns.

A function computing the p-norm $\left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$ of $x \in R^n$:

```
g<-function(x,p=2) {
    #my pnorm function
    y=abs(x)
    z=sum(y^p)
    return(z^(1/p))
}
```

`g(c(3,4))` returns 5 (no `p` set so default `p=2` used).
`g(c(3,4),1)` or `g(c(3,4),p=1)` returns 7.

A function returns the value of the last statement evaluated so the `return()` command is often omitted.

Variable scope

The scope of a variable tells us where the variable is visible. Generally speaking you want a variable to be visible only where it is needed, to avoid clashes with other variables that have the same name.

In R, the variables in a function are local to the function. The environment calling the function (for example, the R-console workspace, or another function) cant see the variables inside the function (only the return values). On the other hand the called function can see the variables in the environment that called it (the parent environment).

For example, if we define a function `f()` in the R-console

```
f<-function(x) {
   a=b*x^2
   return(a)
}
```

set `a=2`, `b=1` and call `f(5)` the function returns 25.

The function found `b` in the workspace that called it.

In the console `a` is still 2 because the function created its own local variable `a`.

`for` loops

A `for`-loop repeats some commands a fixed number of times.

Example: Plan and write a function computing the first n terms in the Fibonacci sequence `1 1 2 3 5 8 13 21 ...`.

We will build up the sequence in a vector $x = (x_1, x_2, ..., x_n)$. In order to calculate the ith entry we will add x_{i-1} and x_{i-2}, the previous two elements of the vector, and write the result into the ith entry in x.

```
fibonacci = function(n) {
    #evalute first n terms in Fibonacci sequence
     x = numeric(n)
     x[1:2] = 1
     for(i in 3:n) {x[i] = x[i-2] + x[i-1]}
     return(x)
}
```

The general form is

```
for (variable in sequence) { statements }
```

`if` and `if else` statements

The `if()` statement controls which statements are executed.

```
if (x > 2) {
    y = 2 * x
} else {
    y = 3 * x
}
```

sets `y=3` when `x = 1` and sets `y=8` when `x=4`.

Can have a simple `if()` without the `else`. The following gives a warning when `x=1` and `y=3` (so `z=NaN`)

```
z=(x-1)/(y-3)
if (is.nan(z)) warning('z is not a number')
```

The `while()` loop

Repeat a set of statements until a condition is satisfied.

Write an R-function to simulate a standard normal random variable conditioned to be greater than a for given real a.

```
Z<-function(a) {
   z=rnorm(1) #rnorm(1) simulates 1 N(0,1) rv
   while (z<a) {
      z=rnorm(1)
   }
   z
}
```

The code repeats the simulation until the condition is satisfied. Now `Z(2)` simulates $Z|Z > 2$ for $Z \sim N(0,1)$.

Returning more than one function output

Use a list to return multiple outputs to the calling environment.

Example: modify the previous example to return the simulated value and the number of trials.

```
Z<-function(a) {
   count=1
   z=rnorm(1)
   while (z<a) {
      z=rnorm(1)
      count=count+1
   }
   return(list(value=z,trials=count))
```

```
}

> Z(3)
$value
[1] 3.224885

$trials
[1] 773
> 1/(1-pnorm(3))
[1] 740.7967
```

The Newton method for root finding

Suppose we wish to find a root of an algebraic equation

$$f(x) = 0.$$

If $f(x)$ has a derivative $f'(x)$, then the following iteration will in general converge to a root if started close enough to the root.

$$\begin{aligned} x_0 &= \text{initial guess} \\ x_n &= x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \end{aligned}$$

The algorithm runs until $|f(x_n)| < \epsilon$.
The idea is based on the Taylor approximation

$$f(x_n) \approx f(x_{n-1}) + (x_n - x_{n-1}) f'(x_{n-1})$$

NOTE : this method may fail to converge.

Newton-Raphson Example

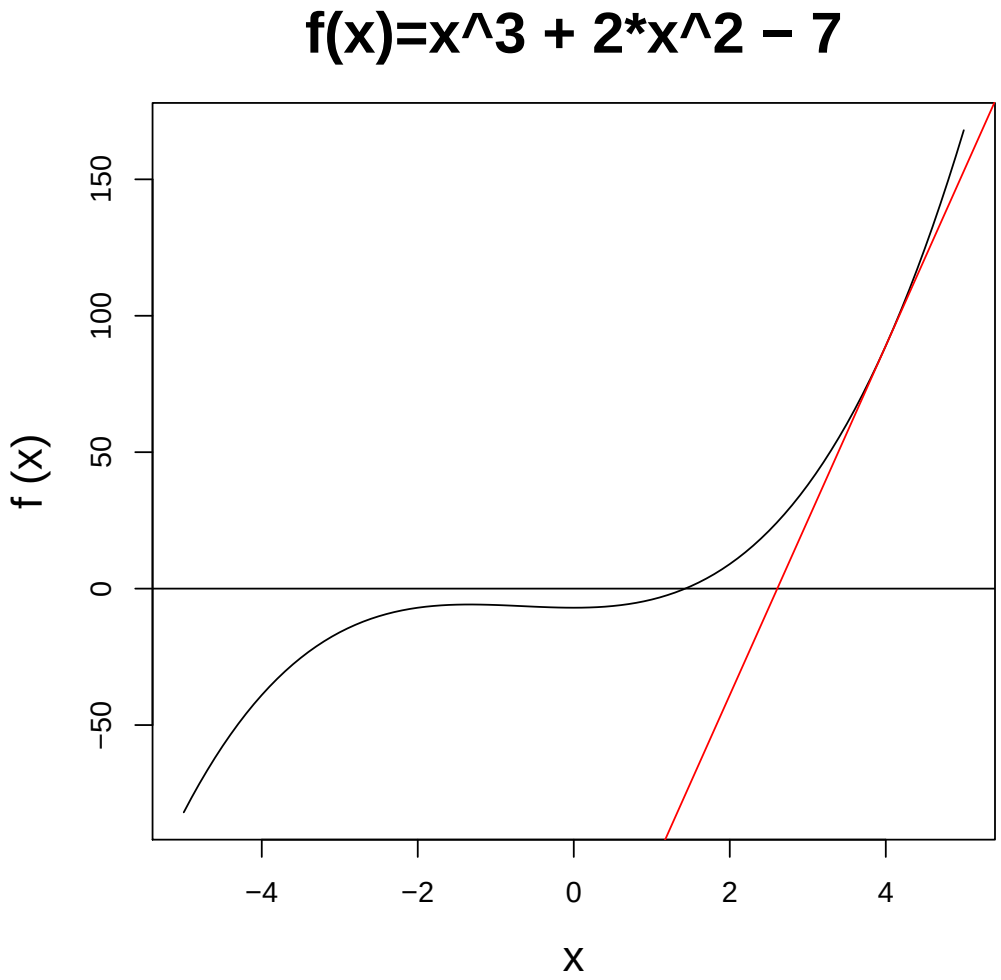
Suppose $f(x) = x^3 + 2x^2 - 7$.

```
f <- function(x) {x^3 + 2*x^2 - 7}
f.prime <- function(x) {3*x^2 + 4*x}

nr <- function(x, tol = 0.001) {
   #Newton-Raphson iteration for f
   while(abs(f(x)) > tol) {
      x = x - (f(x) / f.prime(x))
   }
   return(x)
}
```

The return value for `nr(4)` is approx. 1.428820. Compare 1.428817702 (Maple, numerical evaluation of exact solution to cubic).

Example (first iteration shown starting at $x_0 = 4$).



See you next week

Homework: install R and complete the practical in your own time.