Programming

Robert B. Gramacy
Virginia Tech Department of Statistics
bobby.gramacy.com

Functions

Function objects are defined with this syntax

`function(`*arguments*`)` *body*

where

- *arguments* is a set of symbol names (and, optionally, default values) that will be defined within the body of the function,
- and *body* is an R expression—usually enclosed in curly braces, but that's not necessary for single expressions.

```
> f <- function(x,y) x + y
> f <- function(x,y) { x + y }
```

Arguments and defaults

If you specify a default value for an argument, then it is considered optional.

```
> f(1,2)
[1] 3
> g <- function(x, y=10) { x + y }
> g(1)
[1] 11
> f(1)
Error in x + y : 'y' is missing
> g(1,2)
[1] 3
```

- You will only get an error if you try to use the uninitialized argument within the function body.
- So functions can have optional arguments, if you provide extra checks and conditional execution.

```
> h <- function(x, y) {
+   args <- as.list(match.call())
+   if(is.null(args$y)) {
+     y <- 10
+   }
+   x + y
+ }
> h(1)
[1] 11
```

It is often convenient to specify a variable-length argument list.

- You may want to pass extra arguments to another function,
- or you may want the function to accept a variable number of arguments.

You can do this in R with ellipsis (. . .) in the argument list.

Here is a function that prints the first argument and then passes the other arguments to `summary()`.

```
> f <- function(x, ...) {
+   print(x)
+   summary(...)
+ }
> v <- sqrt(1:100)
> f("Summarizing v:", v, digits=2)
[1] "Summarizing v:"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0     5.1     7.1     6.7     8.7    10.0
```

- All of the arguments after `x` were passed to `summary()`.

You can read arguments from the variable-length list.

```
> addemup <- function(x, ...) {
+   args <- list(...)
+   for(a in args) x <- x + a
+   x
+ }
> addemup(1, 2, 3, 4, 5)
[1] 15
```

- You can refer to items within `...` with `..1`, `..2`, etc.,
- and named arguments become valid symbols within the body of the function.

Return values

Like with many other languages there is a `return()` function to explicitly specify the value being returned.

- It is most useful when the object you wish to return is not the last expression in the function.

```
> f <- function(a, b) {
+   if(a < b) return(a)
+   b
+ }
> f(1,2)
[1] 1
```

Explicit `return()`s may lead to cleaner code.

Functions as arguments

Many functions in R take other functions as arguments.

- `sapply()` is a good example; it iterates through each element of a vector, applying a function to each and returning the results.

```
> a <- 1:7
> sapply(a, sqrt)
[1] 1.000000 1.414214 1.732051 2.000000
[5] 2.236068 2.449490 2.645751
```

- This is a little silly since `sqrt` is already vectorized, as are many built-in functions in R. (see `.R` file)

It is sometimes expedient to create functions without names, i.e., anonymous functions,

- usually to pass them as arguments to other functions.

Here is a simple example.

```
> apply.to.three <- function(f) { f(3) }
> apply.to.three(function(x) { x * 7 })
[1] 21
```

- Essentially, anonymous functions bypass assigning a function object to a symbol.

This is most common with the "apply" family of functions. E.g.,

Instead of

```
> v <- 1:20
> w <- NULL
> for(i in 1:length(v)) { w[i] <- v[i]^2 }
```

the following is a lot clearer, but may not be faster:

```
> w <- sapply(v, function(i) { i^2 })
```

(see `.R` file)

Functions exist to manipulate the behavior of other functions. E.g., you can separately access, and change, the arguments and body of a function.

```
> f <- function(x, y=1, z=2) { x + y + z }
> ff <- formals(f)
> ff$y <- 3
> formals(f) <- ff
> body(f) <- expression({ x * y * z })
> f
function (x, y = 3, z = 2)
{
    x * y * z
}
```

Argument Order and Names

When you specify a function in R

- you assign a name to each argument
- arguments passed into the function are copied from the calling environment
- and inside the function body you can access them by their argument name.

Consider the following simple function.

```
> addTheLog <- function(first, second) {
+   first + log(second)
+ }
```

When calling, you can specify arguments in three ways.

1. Argument order

```
> addTheLog(1, exp(4))
[1] 5
```

2. Exact names (in any order).

```
> addTheLog(second=exp(4), first=1)
[1] 5
```

3. Partial names (in any order).

```
> addTheLog(s=exp(4), f=1)
[1] 5
```

Combinations are allowed ...

- It is common to specify the first few arguments in order, and the latter few (optional) arguments by name, leaving others to defaults.
- Other mixes are confusing.

When calling generic functions, you cannot specify the first argument's name, whose class defines the method.

- E.g., for an `lm` object called `lmo`, do `plot(lmo)` not `plot(x=lmo)`.
- The rest of the arguments can be specified by name.

Side Effects

All functions in R return a value.

Some functions also do other things:

- change variables in the current environment (or in other environments)
- plot graphics,
- load or save files,
- access the network.

Some we've seen already (using `parent.frame()`), and some are the topic of future lectures.

An important function that causes side effects is `<<-`. E.g.,

var `<<-` *value*

It causes the interpreter to first search through the current environment to find `var`. If its not defined then it will look in the parent environment, etc.

- Ultimately creating a new variable called `var` in the global environment if such a symbol is not found.
- Otherwise (re)-assigning to the first `var` symbol it finds the object `value`.

```
> x
Error: object 'x' not found
> doesnt.assign.x <- function(i) x <- i
> doesnt.assign.x(4)
> x
Error: object 'x' not found
> assigns.x <- function(i) x <<- i
> assigns.x(4)
> x
[1] 4
```

Recursion

Recursive programming is a powerful programming technique, made possible by functions.

- A recursive program is simply one that calls itself.
- This is useful because many algorithms and mathematical formulas are recursive in nature.

Consider the Fibonacci sequence:

$$F[1] = F[2] = 1$$
$$F[n] = F[n-1] + F[n-2], \quad n = 3, 4, 5, \ldots$$

Here is one recursive implementation.

```
> fib1 <- function(n, verb=0) {
+   if(verb > 0)
+     cat("called fact1(", n, ")\n", sep="")
+   if(n == 1 || n == 2) return(1)
+   else return(fib1(n-1, verb) + fib1(n-2, verb))
+ }
> fib1(5)
[1] 5
```

But this is inefficient since it duplicates effort.

```
> fib1(5, verb=1)
called fact1(5)
called fact1(4)
called fact1(3)
called fact1(2)
called fact1(1)
called fact1(2)
called fact1(3)
called fact1(2)
called fact1(1)
[1] 5
```

Is there a better way?

How about a less trivial example, where recursion is essential.

The *Sieve of Eratosthenes* is an algorithm for finding all primes less than or equal to a given number n.

Here are the steps.

1. Start with the list $2, 3, \ldots, n$ and the largest known prime $p = 2$.
2. Remove from the list all elements that are multiples of p (but keep p itself).
3. Increase p to the smallest element of the remaining list that is larger than the current p.
4. If p is larger than $\sqrt{n}$ then stop, otherwise go to 2.

```
> primesieve <- function(siev, unsiev, v=0) {
+     if(v > 0) {
+       cat("sieved", siev, "\n")
+       cat("unsieved", unsiev, "\n");
+     }
+     p <- unsiev[1]
+     n <- unsiev[length(unsiev)]
+     if(p^2 > n) return(c(siev, unsiev))
+     else {
+       unsiev <- unsiev[unsiev %% p != 0]
+       siev <- c(siev, p)
+       return(primesieve(siev, unsiev, v))
+     }
+   }
```

```
> primesieve(c(), 2:200)
 [1]   2   3   5   7  11  13  17  19  23  29
[11]  31  37  41  43  47  53  59  61  67  71
[21]  73  79  83  89  97 101 103 107 109 113
[31] 127 131 137 139 149 151 157 163 167 173
[41] 179 181 191 193 197 199
```

- It is a terse but clever program that is faster than many alternatives.

Just to keep the example going ...

Let $\rho(n)$ be the number of primes less than or equal to n. Both Legendre and Gauss famously asserted that

$$\lim_{n \to \infty} \frac{\rho(n) \log(n)}{n} \to 1.$$

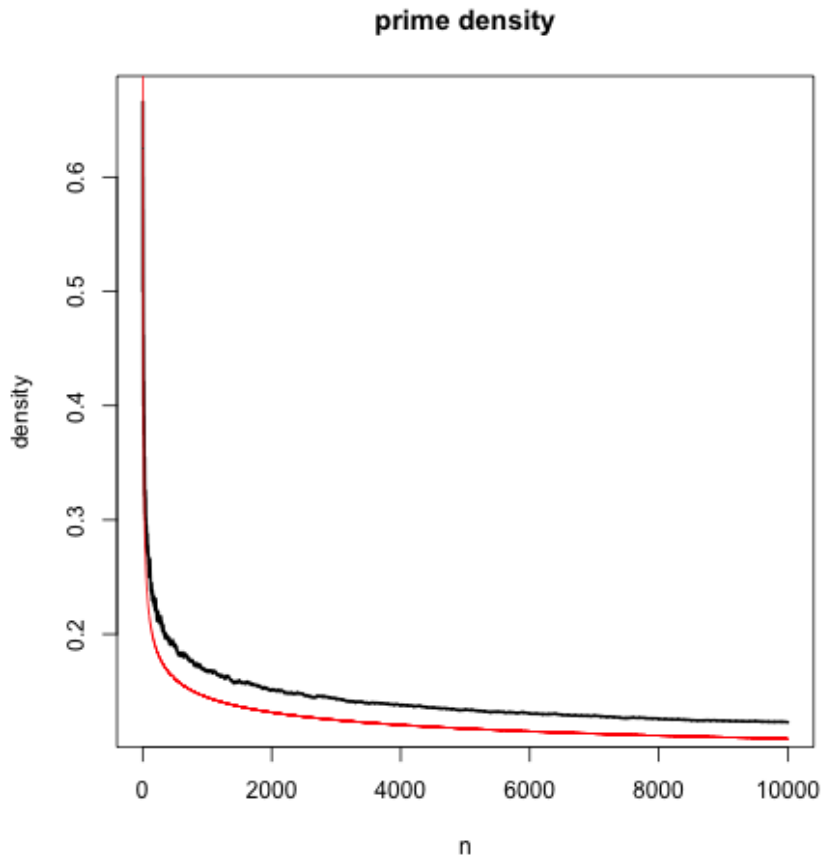
The result was eventually proved some time later by Hadamard and de la Vallée Poussin in 1896.

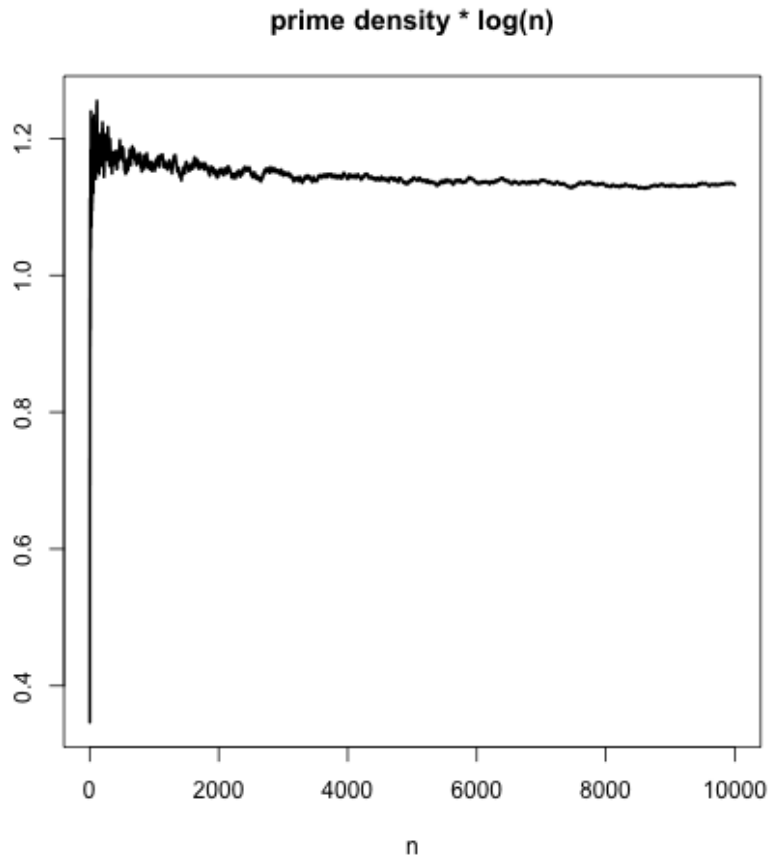
- The proof is hard, but we can easily check the result numerically.

- We'll use our new `primesieve()` and the built-in `cumsum()` function for cumulative sums of a vector.

```
> n <- 10000
> ps <- primesieve(c(), 2:n)
> primes <- rep(0, n-1)
> primes[ps-1] <- 1
> density <- cumsum(primes)/(2:n)

> plot(2:n, density, type="l",
+      main="prime density",
+      xlab="n", ylab="density", lwd=2)
> lines(2:n, 1/log(2:n), col="red")
```






```
> plot(2:n, density*log(2:n), type="l", xlab="n",
+   ylab="", main="prime density * log(n)", lwd=2)
```

Preallocation and Vectorization

R code will be faster if you can reduce the number of times new memory is allocated. E.g.,

```
> seq1 <- function(n) {
+     F <- c(1,1)
+     for(i in 3:n) F[i] <- F[i-1]/F[i-2] + F[i-2]
+     F
+   }
> system.time(s1 <- seq1(100000))
   user  system elapsed
 25.732   5.489  33.091
```

Its way faster if to first create a right-sized `F` so it doesn't have to be re-allocated (automatically) each iteration.

```
> seq2 <- function(n) {
+   F <- rep(NA, n)
+   F[1:2] <- 1
+   for(i in 3:n) F[i] <- F[i-1]/F[i-2] + F[i-2]
+   F
+ }
> system.time(s2 <- seq2(100000))
   user  system elapsed
  0.498   0.000   0.501
```

Its also faster if you can avoid `for` loops altogether. Consider a function summing the first n squares.

```
> s2 <- function(n) {
+   S <- 0
+   for(i in 1:n) S <- S + i^2
+   S
+ }
> system.time(s2(1000000))
   user  system elapsed
  1.028   0.005   1.089
> system.time(sum((1:1000000)^2))
   user  system elapsed
  0.035   0.000   0.036
```

Many of R's functions are vectorized

- which means that if the first argument is a vector, then the output will be a vector of the same length,
- computed by applying the function element-wise.
- Vectorization (usually) allows for faster executing code that is easier to read.
- User-defined functions can also be vectorised if they comprise of functions that are innately vectorised, or are invoked using one of the `apply` family of functions.

Here are a collection of different approaches to solving the problem of summing across the columns of a matrix.

```
> bigM <- matrix(runif(1e+07), nrow=1000)
```

Using a double-`for` loop:

```
> csfor <- function(M) {
+   cs <- rep(NA, ncol(M))
+   for(i in 1:ncol(M)) {
+     s <- 0
+     for(j in 1:nrow(M)) s <- s + M[j, i]
+     cs[i] <- s
+   }
+ }
> system.time(cs1 <- csfor(bigM))
   user  system elapsed
 10.096   0.020  10.166
```

Using `apply()`, which works like `sapply()`, except iterates over rows or columns of a matrix.

```
> system.time(cs2 <- apply(bigM, 2, sum))
   user  system elapsed
  0.276   0.087   0.367
```

Using a single loop of `sum()`s.

```
> system.time({
+  cs3 <- rep(NA, ncol(bigM))
+  for(i in 1:ncol(bigM)) cs3[i] <- sum(bigM[,i])
+ })
   user  system elapsed
  0.218   0.021   0.239
```

Using the built-in `colSums()` function.

```
> system.time(cs4 <- colSums(bigM))
   user  system elapsed
  0.016   0.000   0.016
```

Why is this fastest?

- Because its implemented in C. Most built-in functions are.
- Interpreting R code is much slower than executing compiled C code.

Matrix operations

R's matrix/vector linear algebra routines are state-of-the-art.

- They are linked against to BLAS/Lapack libraries like MATLAB.
- But only the product is in operator form.

Suppose you wish to calculate

$$\hat{\beta} = (X^\top X)^{-1} X^\top y,$$

i.e., the OLS estimator for a linear model.

Here is how we'd do that in R commands.

```
> X <- cbind(1, as.matrix(trees[,1:2]))
> y <- trees[,3]
> XtX <- t(X) %*% X
> XtXi <- solve(XtX)
> XtXiXt <- XtXi %*% t(X)
> XtXiXt %*% y
                [,1]
      -57.9876589
Girth   4.7081605
Height  0.3392512
```

Or, in one expression:

```
> beta.hat <- solve(t(X) %*% X) %*% t(X) %*% y
> beta.hat
                [,1]
      -57.9876589
Girth   4.7081605
Height  0.3392512
```

And just to double-check

```
> coef(lm(y~X[,1]+X[,2]))
(Intercept)      X[, 1]      X[, 2]
-57.9876589   4.7081605   0.3392512
```

Matrix/vector multiplication and inversion are perfect examples of vectorized operations you wouldn't want to re-code.

- An R implementation would be cumbersome.
- Even a bespoke C implementation of specific operations would be folly.

The libraries doing the work are heavily optimized.

- Only a bespoke *compiled* implementation of a long chain of matrix/vector operations has a chance of being faster,
- and only one that also used the same underlying libraries.
- Why? Because R is uses immutable objects.

Object Oriented Programming

At its heart, R is a *functional* programming language.

But it includes some support for OOP, which has become a popular paradigm for organizing computer software.

- In a nutshell, OOP means that objects/data structures are grouped together under a `class` label
- and associated with methods which are functions that operate on objects in the `class`.
- Usually the names of the methods are shared across classes, but do something particular depending on the `class` of the object supplied.

Confusingly, R includes two different mechanisms for OOP, which has to do with the S/S-plus legacy.

- S3-classes, which were introduced in S, version 3.
- S4-classes, from S, version 4.

S4 classes are more sophisticated, complicated, and more *truly* OO.

But S3 are simpler and more common and leverage many of the built-in generic methods we are getting familiar with:

- `plot`, `print`, `summary`, etc.
- So we'll focus on S3.

Establishing a new S3 class is as simple as setting a `class` attribute.

```
> myobj <- list(v=sample((1:20)^2), m="squares")
> class(myobj) <- "myclass"
> myobj
$v
 [1]   1  16 400 289 169 361 324 100  64   9
[11] 225 256  36 196 121  49  25   4 144  81

$m
[1] "squares"

attr(,"class")
[1] "myclass"
```

- Now we have the bare bones needed to start defining methods associated with the class `"myclass"`.

Its easiest to extend existing generic methods. E.g.,

```
> print.myclass <- function(x, ...) {
+   cat("This is a myclass object containing\n")
+   cat("a", class(x$v), "vector ")
+   cat("with", length(x$v), x$m, "\n")
+ }

> myobj
This is a myclass object containing
a numeric vector with 20 squares
```

How does it work?

- Lets look at how `print()` is defined.

```
> print
function (x, ...)
UseMethod("print")
<bytecode: 0x7fcf4476c1c8>
<environment: namespace:base>
```

So when you call the `print()` function, it calls `UseMethod()`, which looks at the `class` attribute of `x` and then (attempts to) call a function named *print.class(x, ...)*.

- The new method's arguments must, at minimum, match those of the generic method.

```
> names(formals(print))
[1] "x"   "..."
```

- You can still inspect the contents of the object.

```
> names(myobj)
[1] "v" "m"
> myobj$v
 [1]   1   4   9  16  25  36  49  64  81 100
[11] 121 144 169 196 225 256 289 324 361 400
```

`plot` and `summary` are other popular generic methods.

```
> plot.myclass <- function(x, ...) {
+   plot(x$v, ylab=x$m, ...)
+ }
> plot(myobj, main="Parabola!")
> summary.myclass <- function(x, ...)
+   {
+     cat("Your", x$m, "are summarized as\n")
+     summary(x$v)
+   }
> summary(myobj)
```

(see `.R` session for output)

- Use the `methods()` function see the methods defined in the current environment for certain generic functions

```
> methods(plot)
...
[13] plot.HoltWinters*   plot.isoreg*
[15] plot.lm             plot.medpolish*
[17] plot.mlm            plot.myclass
[19] plot.ppr*           plot.prcomp*
...
```

- or the methods defined for a particular class

```
> methods(class="myclass")
[1] plot.myclass    print.myclass
[3] summary.myclass
```

You can make your own generic methods too.

```
> penultimate <- function(x)
+     UseMethod("penultimate")
> penultimate.myclass <- function(x) {
+   sort(x$v)[length(x$v)-1]
+ }
> penultimate(myobj)
[1] 361
> penultimate(myobj$v)
Error in UseMethod("penultimate") :
  no applicable method for 'penultimate' applied
  to an object of class "c('double', 'numeric')"
```

- We could fix that!

```
> penultimate.numeric <- function(x) {
+   sort(x)[length(x)-1]
+ }
> penultimate(myobj$v)
[1] 361
> M <- matrix(1:10, nrow=2)
> penultimate(M)
[1] 9
```

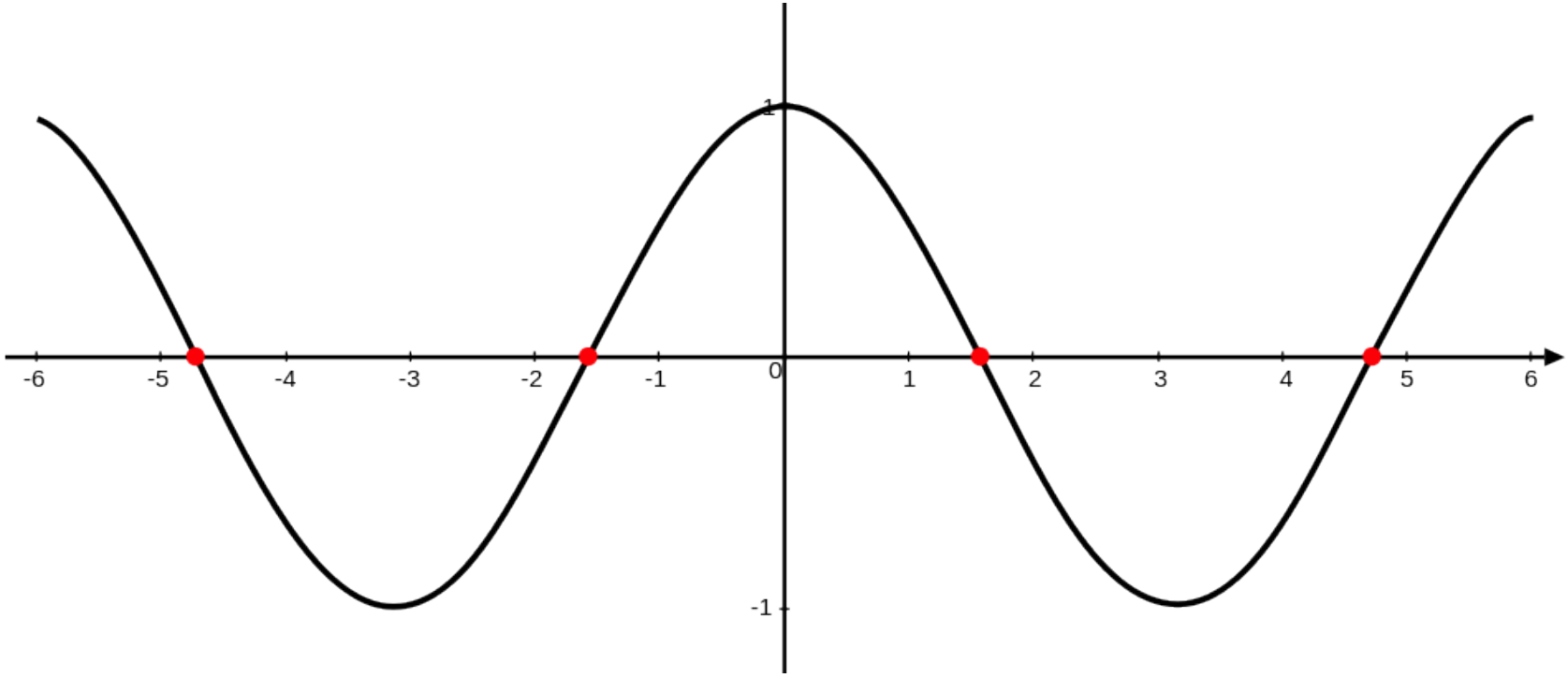
- Notice how coercion allows the generic method for numeric classes to be applied to matrix objects.

An example: root finding by bisection

Suppose that $f : \mathbb{R} \to \mathbb{R}$ is a continuous function.

- A root of f is a solution to the equation $f(x) = 0$.
- That is, a root is a number $a \in \mathbb{R}$ such that $f(a) = 0$.
- If we draw the graph of our function, say $y = f(x)$, which is a curve in the plane, a solution of $f(x) = 0$ is the x-coordinate of a point at which the curve crosses the x-axis.

Roots are important for factorizing polynomials and minimizing functions, and this is not always easy to do analytically.



These are some of the roots of $\cos(x)$.

One of the best *numerical* methods for root finding of 1-d functions $f(x)$ is the bisection method.

If f is a continuous function, then f has a root in the interval (x_l, x_r) if either

- $f(x_l) < 0$ and $f(x_r) > 0$, or
- $f(x_l) > 0$ and $f(x_r) < 0$.

The bisection method works by

- taking an interval (x_l, x_r) that contains a root,
- then successively refining x_l and x_r until $x_r - x_l < \varepsilon$, where ε is some pre-defined tolerance.

Here is the algorithm in pseudocode.

Start with inputs $x_l < x_r$ such that $f(x_l)f(x_r) < 0$, and a desired tolerance ε.

1. If $x_l - x_r < \varepsilon$ then stop.
2. Let $x_m \leftarrow (x_l + x_r)/2$; if $f(x_m) = 0$ stop.
3. If $f(x_l)f(x_m) < 0$ then let $x_r \leftarrow x_m$; otherwise let $x_l \leftarrow x_m$.
4. Goto 1.

Note that at every iteration of the algorithm we know there is a root in the interval (x_l, x_r), so

- it is guaranteed to converge
- with the approximation error reducing by $1/2$ at each iteration.
- If it stops when $x_r - x_l < \varepsilon$, then we know that both x_l and x_r are within distance ε of a root.

(The code, which is included in the accompanying `.R` file, is too long to paste here.)

We'll illustrate the method on

$$f(x) = \log(x) - \exp(-x), \quad x \in (1, 2).$$

```
> f <- function(x) log(x) - exp(-x)
> fr <- bisection(f, 1, 2)
> fr
Root of:
function(x) log(x) - exp(-x)
in (1, 2) found after 27 iterations: 1.3098
to a tolerance of 1.490116e-08
```

- ... we have augmented the generic `print()` method

... and the `summary()` and `plot()` methods too.

```
> plot(fr, ...)
> plot(fr, ..., after=20)
```