Computing Lectures

1 R Basics

Introduction to computing/programming using slides.

Using R as a pocket calculator:

- arithmetic operators (`+`, `-`, `*`, `/`, and `^`)
- elementary functions (`exp` and `log`, `sin` and `cos`)
- `Inf` and `NaN`
- `pi` and `options(digits)`
- comparison operators (`>`, `>=`, `<`, `<=`, `==`, `!=`): informally introduce logicals which will be discussed in more detail in unit 2.

Using variables:

- assignment using `<-` ("gets"): binding symbols to values)
- inspect the binding (show the value of an object) by typing the symbol (name of the object): auto-prints. Show that explicitly calling `print()` has the same effect.
- inspect available bindings (list objects) using `ls()` or `objects()`
- remove bindings ("delete objects") using `rm()`

Using sequences:

- sequences, also known as *vectors*, can be combined using `c()`
- functions `seq()` and `rep()` allow efficient creation of certain patterned sequences; shortcut `:`
- subsequences can be selected by *subscripting* via `[`, using positive (numeric) indices (positions of elements to select) or negative indices (positions of elements to drop)
- can compute with whole sequences: *vectorization*
- summary functions: `sum`, `prod`, `max`, `min`, `range`
- can also have sequences of character strings
- sequences can also have names: use

```
R> x <- c(A = 1, B = 2, C = 3)
R> names(x)

[1] "A" "B" "C"

R> ## Change the names.
R> names(x) <- c("d", "e", "f")
R> x
```

```
d e f
1 2 3
```

```
R> ## Remove the names.
R> names(x) <- NULL
R> x
```

```
[1] 1 2 3
```

for extracting and replacing the names, aka as *getting* and *setting*

Using functions:

- start with the simple standard example

  ```
  R> hell <- function() writeLines("Hello world.")
  ```

  to explain the constituents of functions: argument list (formals), body, and environment. Invocation via

  ```
  R> hell()
  ```

  ```
  Hello world.
  ```

- Introduce basic programming elements: grouping and flow control (conditionals and loops).
- Use e.g.

  ```
  R> for(i in 1 : pi) cat(i, "\n")
  ```

  ```
  1
  2
  3
  ```

  to introduce `cat()` and the `"\n"` for the newline, deferring more detailed discussion of string constants to unit 2.

- Discuss return value of functions, noting that

  ```
  R> val <- hell()
  ```

  ```
  Hello world.
  ```

  ```
  R> val
  ```

  ```
  NULL
  ```

- Use this to introduce `NULL` ("nothing").
- Discuss the difference between *outputting* something and *returning* something.
- Discuss fundamental principles:

  1. Everything in R is an object
  2. All computations are function calls which have a value
  3. Some function calls are performed for their *side effects*

Using I/O:

- Discuss I/O basics: R reads input from an input connection (typically, `stdin()`) and writes output and messages to an output and message connection, respectively (typically, `stdout()` and `stderr()`. The latter two can be redirected using `sink()`.
- R code in a file can be read using `source()`
- Textual representations of R objects (appropriate for reading in via `source()` can be created using `dump()`
- Objects ("data") can also be saved and loaded in portable machine formats using `save()` and `load()`.
- Discuss interaction styles and persistence: editing text files with code and using "inferior" R process versus "typing at the prompt" (and e.g. saving transcripts)
- Discuss text editors versus word processors
- Explain submitting homework assignments: send R code as plain text attachments

Getting help:

- function `help()` and question-mark shortcut; mention `help.search()`
- `help.start()` for bringing up a web interface with direct access to the manuals.

R environment:

- Discuss (G)UIs and IDEs
- R-aware text editing (indentation, highlighting, ...)
- Interaction with the editor
- Interaction with the help system
- Transscripts

2 R Data Types

2.1 NULL

2.2 Logicals

Logical constants: `TRUE` and `FALSE` ... and `NA`.

Discuss: 3-valued logic, missingness.

Logical vectors can be obtained by combining logical constants; by creation via `logical()` (or `vector(mode = "logical")`), or by coercion via `as.logical()`. Note

```
R> logical(3)

[1] FALSE FALSE FALSE

R> logical()
```

```
logical(0)
```

Discuss the difference between an empty sequence (zero-length vector) of a certain mode and `NULL` (which is really nothing).

```
R> typeof(logical())

[1] "logical"

R> length(logical())

[1] 0
```

Also obtained as the results of comparisons.

Computational model: *not* single bits as 3-valued logic. (In fact, single bytes are used, which is very inefficient.)

Basic functions/operators: negation (`!`), and and or (`&` and `|`), comparisons (in particular, `==` and `!=`). Note the truth tables:

```
R> x <- c(TRUE, FALSE, NA)
R> x

[1]  TRUE FALSE    NA

R> !x

[1] FALSE  TRUE    NA

R> outer(x, x, `&`)

      [,1]  [,2]  [,3]
[1,]  TRUE FALSE    NA
[2,] FALSE FALSE FALSE
[3,]    NA FALSE    NA

R> outer(x, x, `|`)

     [,1]  [,2] [,3]
[1,] TRUE  TRUE TRUE
[2,] TRUE FALSE   NA
[3,] TRUE    NA   NA

R> outer(x, x, `==`)

      [,1]  [,2] [,3]
[1,]  TRUE FALSE   NA
[2,] FALSE  TRUE   NA
[3,]    NA    NA   NA
```

Note in particular that we cannot test for missingness using `==`: use `is.na` instead.

Short-circuit operators `&&` and `||` to be used where a single (non-missing) truth value is needed (flow control).

Summary operators: `all` and `any` (and can use `sum` for counting trues).

Recycling: for example,

```
R> x & FALSE

[1] FALSE FALSE FALSE
```

General principle for many functions: arguments are recycled to a common length (with a warning if these resulted in fractional multiples).

Subscripting using logicals: select elements for which subscript is true (recycled as needed).

2.3 Integers

Integer constants: numeric constants with a trailing `L`. For example:

```
R> c(32L, 0xffL)

[1]  32 255
```

(using decimal and hexadecimal constants).

`NA_integer_`.

Integer vectors can be obtained by combining integer constants; by creation via `integer()` (or `vector(mode = "integer")`), or by coercion via `as.integer()`.

In addition, the various rounding operations (`floor` and `ceiling`, `trunc` and `round(digits = 0)`) result in integers. Note

```
R> round(c(0.5, 1.5, 2.5, 3.5))

[1] 0 2 2 4
```

According to the IEC 60559 standard, rounding .5 is by going to the even digit.

Integers are also obtained from `length()` and certain `seq()` invocations.

Computational model: signed 32 bits (4 bytes) (C data type `int`, not `long int` which is 64 bits on 64 bit platforms). Different schemes to encode integers (e.g., `http://en.wikipedia.org/wiki/Signed_number_representations`):

- In the sign-and-magnitude scheme, one (e.g., the leading) bit is used for the sign, and using k bits (for R, $k = 32$)

  $$\pm b_{k-2} \cdots b_1 b_0 \leftrightarrow \pm(b_{k-2} 2^{k-2} + \cdots + b_1 2^1 + b_0 2^0).$$

  Disadvantage: two representations of 0 (one of which could be used to indicate missingness).

- In the biased scheme,

  $$b_{k-1} \cdots b_1 b_0 \leftrightarrow (b_{k-1} 2^{k-1} + \cdots + b_1 2^1 + b_0 2^0) - (2^{k-1} - 1)$$

  Disadvantage: addition becomes a bit complex.

- In the most popular *two's complement* scheme (e.g., `http://en.wikipedia.org/wiki/Two%27s_complement`):

  - $0, 1, \ldots, 2^{k-1} - 1$ are represented the usual way;
  - $-1, -2, \ldots, -2^{k-1}$ are represented by (bit sequences giving the binary representation of) $2^k - 1$, $2^k - 2$, ..., $2^k - 2^{k-1} = 2^{k-1}$.

  Or, more compactly: the value corresponding to the bit sequence $b_{k-1} \cdots b_0$ is $-b_{k-1} 2^{k-1} + \sum_{i=0}^{k-2} b_i 2^i$.

  In R, the smallest possible integer (-2^{31}) is then taken as `NA_integer_`: corresponds to bit pattern $10 \cdots 0$ or 0x80000000.

What is the largest possible integer? Using the simple sign-and-magnitude scheme and taking all bits one leaves

$$2^0 + 2^1 + \cdots + 2^{30} = \frac{2^{31} - 1}{2 - 1} = 2^{31} - 1.$$

for the magnitude, as for two's complement.

Note

```
R> .Machine$integer.max

[1] 2147483647

R> as.integer(2^31 - 1)

[1] 2147483647

R> as.integer(2^31)

[1] NA
```

2.4 Doubles

Numeric constants are similar to C: they consist of an integer part consisting of zero or more digits, followed optionally by '`.`' and a fractional part of zero or more digits optionally followed by an exponent part consisting of an '`E`' or an '`e`', an optional sign and a string of zero or more digits. Either the fractional or the decimal part can be empty, but not both at once. Valid numeric constants:

```
R> c(1, 10, 0.1, .2, 1e-7, 1.2e+7)

[1] 1.0e+00 1.0e+01 1.0e-01 2.0e-01 1.0e-07 1.2e+07
```

Numeric constants can also be hexadecimal.

Note that R uses the term "numeric" to refer to numeric objects, but not consistently: `numeric` and `as.numeric` identify numeric with double; `is.numeric` gives true for both doubles and integers. Ignoring the latter, let us conceptually identify numeric and double.

Double vectors can be obtained by combining numeric constants; by creation via `double()` or `numeric()` (or `vector(mode = "numeric")`), or by coercion via `as.double()` or `as.numeric()`.

To make matters even more complicated: `NA_real_`.

Doubles are floating-point numbers following the double precision (64-bit) IEC 60559 (aka IEEE 754) standard.

In general, a *floating-point number system* is characterized by four integers: b (base or radix), p (precision), $e_{\min}$ and $e_{\max}$ (minimal and maximal exponent) and consists of numbers of the form

$$x = \pm \left(d_0 + \frac{d_1}{b} + \cdots + \frac{d_{p-1}}{b^{p-1}} \right) b^e, \qquad 0 \le d_i \le b-1, i = 0, \ldots, p-1; \quad e_{\min} \le e \le e_{\max},$$

where $d_0 \cdots d_{p-1}$ is the *mantissa* (or *significand*) and e is the *exponent* (or *characteristic*) or the floating-point number.

A non-zero number is *normalized* if the leading digit d_0 is non-zero.

For IEEE 754, $b = 2$, $p = 53$, $e_{\min} = -1022$ and $e_{\max} = 1023$, using a 64-bit (8-byte) representation with 1 bit for the sign, 52 bits for the mantissa, and 11 bits for the exponent.

Exponents different from all-zero and all-one correspond to normalized numbers, and are interpreted using the biased scheme. I.e.,

$$e_{10} \cdots e_1 e_0 \leftrightarrow e_{10} * 2^{10} + \cdots e_1 * 2 e_0 - 1023$$

so that indeed

$$e_{\min} = (0 * 2^{10} + \cdots + 0 * 2 + 1) - 1023 = -1022$$

and

$$e_{\max} = (1 * 2^{10} + \cdots + 1 * 2 + 0) - 1023 = 1023.$$

Note that as $b = 2$, for a normalized number $d_0 \equiv 1$, so the redundant d_0 does not need to be stored: only $p-1$ bits are needed for a precision of p by using only the *fraction* $d_1 \cdots d_{p-1}$. Thus, for normalized numbers

$$\sigma | d_1 \cdots d_{52} | e_{10} \cdots e_0 \leftrightarrow \sigma \, 1.d_1 \ldots d_{52} * 2^e = \sigma \left(1 + \frac{d_1}{2} + \cdots + \frac{d_{52}}{2^{52}} \right) 2^e.$$

Exponents with only zeros are used for *denormalized* numbers, with interpretation

$$\sigma | d_1 \cdots d_{52} | 0 \cdots 0 \leftrightarrow \sigma \, 0.d_1 \ldots d_{52} * 2^{-1022} = \sigma \left(\frac{d_1}{2} + \cdots + \frac{d_{52}}{2^{52}} \right) 2^{-1022}.$$

Note that this gives *two* (signed) zeros!

Exponents with only ones are used for representing signed infinities (with all d_i zero) and NaNs.

What is the largest positive number representable? Take $e = 1023$ and all mantissa bits 1 to get

$$\left(1 + \frac{1}{2} + \cdots + \frac{1}{2^{52}}\right) 2^{1023} = 2^{-52}(2^{53} - 1)2^{1023} = 2^{1023}(2 - 2^{-52}).$$

What is the smallest positive number representable? This must be the denormalized number with only the last mantissa bit 1, giving

$$\left(0 + 0 + \cdots + 0 + \frac{1}{2^{52}}\right) 2^{-1022} = 2^{-1074}.$$

Consequences of using floating point arithmetic:

- Most numbers cannot be represented exactly.
- Computations can only use up to 52 mantissa bits. Numbers which would differ in a mantissa bit from 53 upwards are "the same".

  ```
  R> 1 == (1 + 2^(-52))

  [1] FALSE

  R> 1 == (1 + 2^(-53))

  [1] TRUE
  ```

  As

  ```
  R> c(2^(-52), 2^1023)

  [1]  2.220446e-16 8.988466e+307
  ```

  double precision is roughly equivalent to 16 significant decimal figures, with exponents up to size ± 308.

- In a way, "basically nothing" works as you are used to! I.e., the fundamental laws of arithmetic do not hold (exactly).

  E.g., algebraically

  $$x * n/x - n = 0$$

  but numerically:

  ```
  R> x <- 1.25
  R> y <- 1 / x
  R> n <- 1 : 10
  R> x * (n * y) - n

   [1] 0.000000e+00 0.000000e+00 4.440892e-16 0.000000e+00 0.000000e+00
   [6] 8.881784e-16 8.881784e-16 0.000000e+00 0.000000e+00 0.000000e+00
  ```

Or, more dramatically, illustrate "catastrophic cancellation" of rounding errors accumulating in computations. Traditionally, one recommends to use the identity

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right)$$

to "simplify" computation of the sample variance of a sample $x_1, \ldots, x_n$ ("one-pass formula"). Consider

```
R> x <- 1 : 11
R> var(x)

[1] 11
```

Using

```
R> V <- function(x) {
+     n <- length(x)
+     (sum(x ^ 2) - n * mean(x) ^ 2) / (n - 1)
+ }
```

we get

```
R> V(x)

[1] 11

R> V(x + 1e10)

[1] 0
```

due to a catastrophic loss in precision.

2.5 Complex Numbers

Complex constants: decimal numeric constant followed by `i`. (Notice that only purely imaginary numbers are actual constants.) Valid complex constants: `2i`, `4.1i`, `1e-2i`. `NA_complex_`.

Creation and coercion via `complex()` and `as.complex()`.

Also obtained as: `sqrt()` of complex numbers (but not of reals), note:

```
R> sqrt(-1)

[1] NaN

R> sqrt(-1+0i)

[1] 0+1i
```

and by `polyroot()` which finds the (possibly complex) roots of polynomials.

Basic functions: `Re`, `Im`, `Mod`, `Arg`, and `Conj` for the real and imaginary parts, modulus and argument, and the complex conjugate.

2.6 Strings

String constants: are delimited by a pair of double (or single) quotes and can contain all other printable characters. Quotes and other special characters are specified using (backslash) *escape sequences*:

\'	single quote
\"	double quote
\n	newline
\r	carriage return
\t	tab character
\b	backspace
\a	bell
\f	form feed
\v	vertical tab
\\	backslash
NNN	character with given octal code
\x*NN*	character with given hex code
\u*NNNN*	Unicode character with given hex code
\U*NNNNNNNN*	Unicode character with given hex code

`NA_character_`.

Creation and coercion via `character` and `as.character`.

Computational model: tricky (bytes versus (wide) characters). Example: do umlaut-a in a UTF-8 locale:

```
R> sessionInfo()

R version 4.0.2 Patched (2020-06-22 r78735)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Debian GNU/Linux bullseye/sid

Matrix products: default
BLAS:   /usr/local/lib/R/lib/libRblas.so
LAPACK: /usr/local/lib/R/lib/libRlapack.so

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
[1] compiler_4.0.2 tools_4.0.2

R> Sys.getlocale("LC_CTYPE")

[1] "en_US.UTF-8"

R> ula <- "ä"
R> ula

[1] "ä"
```

```
R> nchar(ula)

[1] 1

R> nchar(ula, type = "bytes")

[1] 2

R> charToRaw(ula)

[1] c3 a4
```

Slides for computations on character vectors.

Subscripting via character vectors for objects which have names:

```
R> x <- c(A = 1, B = 2, C = 3)
R> names(x)

[1] "A" "B" "C"

R> x[c("A", "B")]

A B
1 2
```

2.7 Lists

Lists are "generic vectors": sequences of elements of arbitrary mode.

In particular, elements can be lists themselves: *recursive* data type as opposed to *atomic* ones (as discussed thus far).

Creation via `list()` or `vector(mode = "list")`; coercion via `as.list()`.

Flattening out via `unlist()`.

Subscripting:

- `[` extracts sub-lists (using subscripting rules as previously discussed)
- `[[` and `$` extract single elements (as indicated by their position or name)

`lapply()` applies functions to the elements of a list, returning a list with the respective values.

2.8 More general-purpose functions for vectors

In addition to `length()` and subscripting:

`unique()` and `duplicated()` extract unique values or indicate the positions of duplicated elements.

`match()` matches sequence elements.

2.9 Attributes and associated data types

Objects can have attributes:

- *implicit* (type, mode, length)
- *explicit* as lists of attributes ("property lists") which are handled via the `attr()` and `attributes()` getters and setters. Example:

```
R> x <- c(A = 1, B = 2, C = 3)
R> attributes(x)

$names
[1] "A" "B" "C"
```

Some data types are simply objects with certain attributes ("structures"): e.g., matrices and arrays have a `dim` attribute (and may have a `dimnames` attribute.

Creation of matrices via `matrix()`, `rbind()` and `cbind()`; `diag()` for creating diagonal matrices.

```
R> x <- c(1, 2)
R> y <- c(2, 3)
R> rbind(x)

  [,1] [,2]
x    1    2

R> rbind(x, y)

  [,1] [,2]
x    1    2
y    2    3

R> cbind(x, y)

     x y
[1,] 1 2
[2,] 2 3

R> matrix(1 : 4, 2, 2)

     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Note that `rbind()` and `cbind()` add dimnames, and that by default matrices are created in column major order. Subscripting and dimnames will be discussed in unit 6.

Some data types are characterized by their `class` attribute (simple S3 object model): e.g., factors and data frames (fundamental for data analysis, hence discussed in more detail in Stats course).

3 R Programming

3.1 Programming

Functions:

- First-class objects ("everything is an object")
- Can have functions as arguments and/or return values (functional programming; higher-order functions)
- Discuss default values for arguments
- Return values (always a single object, but could be `NULL` for nothing, or a list of objects)
- Named versus positional arguments, mention `...` arguments
- Briefly recall that functions are "closures" which also have an environment, which matters for scoping. R has lexical scoping allowing for arbitrarily nested function declarations.

Programming Elements:

Blocks grouping using curly braces (`{` and `}`)

Conditionals `if`/`else`; mention `switch`

Loops `for` and `while`/`repeat` as basic constructs; `next` advances to the next iteration, `break` terminates the loop.

Task: Find the first n Fibonacci numbers. The Fibonacci numbers are defined by the second order recursion

$$f_1 = f_2 = 1, \qquad f_n = f_{n-1} + f_{n-2}, \quad n > 2.$$

Straightforward implementation using a `for` loop:

```
R> Fib1 <- function(n) {
+     if(n == 1) return(1)
+     fib <- numeric(n)
+     fib[1] <- fib[2] <- 1
+     for(i in seq(3, length.out = n - 2))
+         fib[i] <- fib[i - 1] + fib[i - 2]
+     fib
+ }
```

Note that we allocate storage first to avoid repeated re-allocation.

Task: Find the primes $\leq n$. Use an Eratosthenes-type sieve almost as in the textbook (actually, basically Exercise 2 on page 58 of the book).

```
R> Erato1 <- function(n) {
+     if(n <= 1) return(numeric())
+     sieve <- 2 : n
```

```
+     primes <- numeric()
+     while(length(sieve)) {
+         p <- sieve[1]
+         primes <- c(primes, p)
+         sieve <- sieve[(sieve %% p) != 0]
+     }
+     primes
+ }
```

Task: Find the Fibonacci numbers $\leq n$. Use a function version of the code in the textbook (modulo errors).

```
R> Fib2 <- function(n = 300) {
+     u <- 1
+     v <- 1
+     ## This is wrong in the book.
+     ##   fib <- c(u, v)
+     fib <- u
+     while(v < n) {
+         fib <- c(fib, v)
+         w <- u + v
+         u <- v
+         v <- w
+     }
+     fib
+ }
```

Note that in general iteratively adding a single element to a vector should be avoided, as this keeps reallocating memory. Given the exponential growth of f_n, this effect can be neglected for the above.

Task: Functional version of Newton's method. Newton's method for finding roots of a differentiable function f is based on the recursion

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

obtained by using the first-order Taylor expansion

$$f(x) \approx g_n(x) = f(x_n) + (x - x_n)f'(x_n)$$

and solving $g_n(x) = 0$.

A simple "functional" (taking `f` and `fprime` function arguments, as opposed to using expressions as in the textbook) version is:

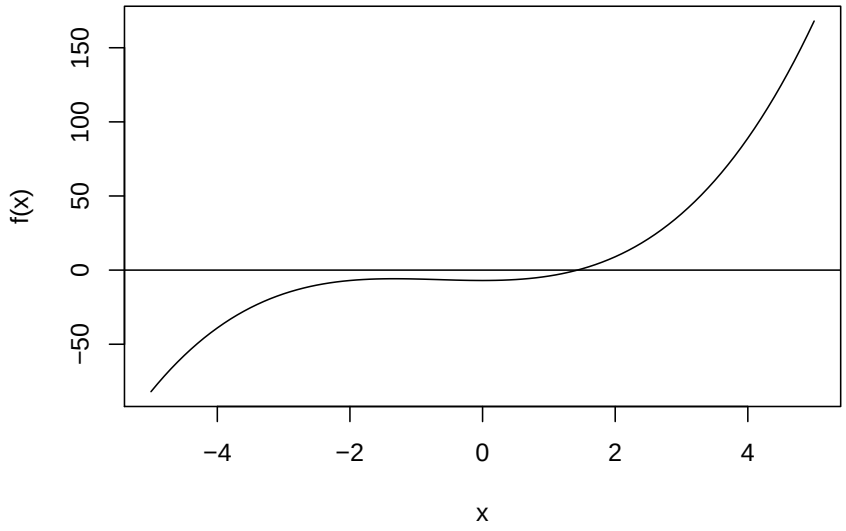
```
R> newton1 <- function(x, f, fprime, tol = 1e-6) {
+     y <- f(x)
+     while(abs(y) > tol) {
+         x <- x - y / fprime(x)
+         y <- f(x)
+     }
+     x
+ }
```

Consider the function

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

This has a single real root:

```
R> f <- function(x) x^3 + 2 * x^2 -7
R> curve(f, from = -5, to = 5)
R> abline(h = 0)
```



All roots:

```
R> polyroot(c(-7, 0, 2, 1))
```

```
[1]  1.428818+0.000000i -1.714409+1.399985i -1.714409-1.399985i
```

We can find the real root using Newton's method:

```
R> fprime <- function(x) 3 * x^2 + 4 * x
R> newton1(1, f, fprime)
```

```
[1] 1.428818
```

Experiment with different starting values to show that this example is particularly well-behaved.

```
R> newton1(10, f, fprime)
```

```
[1] 1.428818
```

Improvements.

What if we want to know the number of iterations?

```
R> newton1a <- function(x, f, fprime, tol = 1e-6, verbose = FALSE) {
+     n <- 0
+     y <- f(x)
+     while(abs(y) > tol) {
+         n <- n + 1
+         if(verbose)
+             cat("Iteration:", n, "\n")
+         x <- x - y / fprime(x)
+         y <- f(x)
+     }
+     x
+ }
R> newton1a(10, f, fprime, verbose = TRUE)

Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
Iteration: 6
Iteration: 7
Iteration: 8
[1] 1.428818
```

What if we want to use the number of iterations in subsequent computations?

```
R> newton1b <- function(x, f, fprime, tol = 1e-6, verbose = FALSE) {
+     n <- 0
+     y <- f(x)
+     while(abs(y) > tol) {
+         n <- n + 1
+         if(verbose)
+             cat("Iteration:", n, "\n")
+         x <- x - y / fprime(x)
+         y <- f(x)
+     }
+     list(x = x, n = n)
+ }
R> newton1b(10, f, fprime, verbose = TRUE)

Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
Iteration: 6
Iteration: 7
Iteration: 8
$x
[1] 1.428818

$n
[1] 8
```

What if we also want to be able to specify a maximal number of iterations?

```
R> newton1c <-
+ function(x, f, fprime, tol = 1e-6, verbose = FALSE, maxiter = 100) {
+     n <- 0
+     y <- f(x)
+     while((abs(y) > tol) && (n < maxiter)) {
+         n <- n + 1
+         if(verbose)
+             cat("Iteration:", n, "\n")
+         x <- x - y / fprime(x)
+         y <- f(x)
+     }
+     list(x = x, n = n)
+ }
R> newton1c(10, f, fprime, verbose = TRUE)

Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
Iteration: 6
Iteration: 7
Iteration: 8
$x
[1] 1.428818

$n
[1] 8
```

Finally, what if we do not want to specify the derivative by hand? See unit 5.

3.2 Debugging

Consider the coefficient of variation example from the textbook.

```
R> CV <- function(x) sd(x / mean(x))
```

Unfortunately, the example in the book no longer fails for current versions of R:

```
R> CV(0)

[1] NA
```

so there is nothing to fix (the variance functions were changed to return `NA` rather than an error in case of insufficient data).

We construct a simple toy problem:

```
R> check_for_large_CV <- function(x, t = 3) {
+     cv <- CV(x)
+     if(cv > t) writeLines(sprintf("CV is %g > 3", cv))
+ }
```

This works fine for $n > 1$ data points:

```
R> check_for_large_CV(1 : 100)
R> check_for_large_CV(1 : 2)
```

but fails for sample size one:

```
R> check_for_large_CV(1)

Error in if (cv > t) writeLines(sprintf("CV is %g > 3", cv)) :
  missing value where TRUE/FALSE needed
```

Where did the error occur? (If we did not know.)

```
R> traceback()

1: check_for_large_CV(1)
```

So the problem is in line

```
    if(cv > t) writeLines(sprintf("CV is %g > 3", cv))
```

in function `check_for_large_CV`: as this is a very short function, we can use `debug()` and step through.

3.3 Profiling

Timings to illustrate the effects of re-allocation and vectorization:

```
R> n <- 10000
R> x <- rnorm(n)
R> y <- rnorm(n)

R> system.time({
+     z1 <- numeric()
+     for(i in seq_len(n)) z1 <- c(z1, x[i] + y[i])
+ })

   user  system elapsed
  0.260   0.024   0.284

R> system.time({
+     z2 <- numeric(n)
+     for(i in seq_along(x)) z2[i] <-  x[i] + y[i]
+ })

   user  system elapsed
  0.012   0.000   0.011

R> system.time(z3 <- x + y)
```

```
   user  system elapsed
      0       0       0
```

One can use `Rprof()` and `summaryRprof()` to profile the execution of R expression: does not work for the above because no events get recorded.

3.4 Recursion

Task: Recursive computation of digit sum. Mimick how we would do this by hand: adding e.g. from right to left can be thought as adding the last digit to the sum of the others. If `n` is a non-zero natural (decimal) number, `n %% 10` extracts the last digit and `n %/% 10` the others, hence:

```
R> digisum <- function(n) {
+     if(n == 0) return(0)
+     (n %% 10) + Recall(n %/% 10)
+ }
```

Task: Recursive version of Eratosthenes sieve. Mimick how we would do this by hand:

- Take all numbers from 2 to n as the first candidate list (sieve).
- Recursively take the first element in the list as the next prime, and eliminate all its multiples from the candidate list.
- Stop recursion when there are no more candidates.

```
R> Erato2 <- function(n) {
+     ## Use a helper function for the recursion
+     helper <- function(sieve) {
+         if(!length(sieve)) return(integer())
+         prime <- sieve[1]
+         sieve <- sieve[(sieve %% prime) != 0]
+         c(prime, Recall(sieve))
+     }
+     ## Call the helper with the initial sieve.
+     helper(seq(2, length.out = n - 1))
+ }
```

Task: Recursive versions of finding the n-th Fibonacci number. (Aka the good and the bad, but no ugly.) An apparently straightforward recursive implementation of the Fibonacci recursion is

```
R> FibR1 <- function(n) {
+     if((n == 1) || (n == 2)) return(1)
+     Recall(n - 1) + Recall(n - 2)
+ }
```

This has very poor time complexity:

```
R> system.time(f25R1 <- FibR1(25))

   user  system elapsed
  0.205   0.001   0.205
```

```
R> f25R1

[1] 75025
```

In fact, if c_n is the number of operations needed to compute f_n, then basically $c_n = c_{n-1} + c_{n-2}$ and hence c_n grows at the same exponential rate as the Fibonacci numbers themselves!

We can do much better by using recursion to compute the whole sequence (and maybe only keep the last element):

```
R> FibR2 <- function(n) {
+     if(n == 1) return(1)
+     if(n == 2) return(c(1, 1))
+     s <- Recall(n - 1)
+     c(s, s[n - 1] + s[n - 2])
+ }
```

This has dramatically better run time complexity

```
R> system.time(f25R2 <- FibR2(25))

   user  system elapsed
  0.008   0.000   0.008

R> f25R2

 [1]     1     1     2     3     5     8    13    21    34    55    89   144
[13]   233   377   610   987  1597  2584  4181  6765 10946 17711 28657 46368
[25] 75025
```

but is still slower than the explicit loop variant. Discuss why.

To make the second-order recursion work, we need to remember the results we already computed ("memoization"). In R, we can use a cache in the environment of the function:

```
R> FibR3 <- local({
+     .cache <- c(1, 1)
+     function(n) {
+         if(n <= length(.cache))
+             .cache[n]
+         else {
+             f <- Recall(n - 1) + Recall(n - 2)
+             .cache[n] <<- f
+             f
+         }
+     }
+ })
```

This performs much better:

```
R> system.time(f25R3 <- FibR3(25))

   user  system elapsed
      0       0       0
```

3.5 Functional Programming

Use `lapply` to apply a function to the elements of a (single) list or vector. Discuss that this is more efficient than using a for loop and collecting results by hand.

```
R> n <- 10000
R> x <- runif(n)
R> system.time({
+     y1 <- numeric()
+     for(i in seq_len(n)) y1 <- c(y1, x[i]^2)
+ })

   user  system elapsed
  0.175   0.015   0.191
```

We know that pre-allocating memory is much better:

```
R> system.time({
+     y2 <- numeric(n)
+     for(i in seq_len(n)) y2[i] <- x[i]^2
+ })

   user  system elapsed
  0.005   0.000   0.005
```

But it is still better to let `lapply` create the result.

```
R> system.time(y3 <- unlist(lapply(x, function(u) u^2)))

   user  system elapsed
   0.01    0.00    0.01
```

Of course, we should really do

```
R> system.time(y4 <- x ^ 2)

   user  system elapsed
  0.001   0.000   0.000
```

Use `mapply` to apply a function to the corresponding elements of multiple lists or vectors.

Use `Reduce` to use a binary function to successively combine the elements of a given vector. See `?Reduce` for functional versions of function iteration based on `Reduce`.

4 LaTeX, BibTeX and Sweave

4.1 LaTeX

History of TeX and LaTeX

Logical versus physical markup

Basic structure: header (preamble) with documentclass, document.

Writing an article: author, title, date; `\maketitle`.

Sectioning and paragraphs.

Environments: itemize, enumerate, description; tabular and tabbing; theorem etc.

Markup: `\emph`. Is explicit font or size manipulation really needed for logical markup?

Special insertions: special characters; dots; space after period; accents.

Math: illustrate via e.g.

$$\frac{1}{2\pi}\int_{-\infty}^{\infty}\left(\sum_{n=0}^{\infty}(-1)^n\frac{x^{2n}}{2^n n!}\right)dx + e^{\sqrt{-1}\pi} = 0.$$

Cross-references: `\label`, `\ref` and `\pageref`.

Floats: figure and table.

Hyperlinks: `\usepackage{hyperref}`.

Including graphics via `\usepackage{graphics}` and `\includegraphics`.

Writing slides: e.g. simple documentclass `slides`; much more advanced systems (e.g., texpower).

4.2 BibTeX

Collect bibliographic information into simple plain text databases (.bib) files. Can be edited by hand, but many information resources can provide bibinfo in BibTeX format:

- obtaining bibentries from CIS
- obtaining bibentries from Wikipedia
- obtaining bibentries from SSRN
- obtaining bibentries from JBF

Integrating info via `\cite` for citations, `\bibliographystyle` and `\bibliography`.

Recommendations:

- use a author-year bibliography style (e.g., plainnat)
- `\usepackage{natbib}` and use `\citet` and `\citep` for textual and parenthetical citations

4.3 Sweave

Integrating R and LaTeX via Sweave.

Discuss integrated text documents and literate programming

Discuss Sweave idea: put R code into code chunks; Sweave will then suitably evaluate these (as well as in-line `\Sexpr`) and show the results. I.e., .Rnw → .tex via Sweave.

Stangle can extract the code in the code chunks.

Examples: basic in-place output; tabular output in tables; figures.

Mention that more fancy output can be generated by having R create LaTeX output and using `results=tex` on the code chunks.

Discuss benefits and usage scenarios.

5 Optimization and Root Finding

Already discussed fixed point and Newton-Raphson methods for finding roots of a univariate function, and mentioned `polyroot()` for finding the n (possibly complex) roots of a degree n polynomial.

One-dimensional root finding can also be performed using the built-in function `uniroot()`.

Obviously, solving $f(x) = 0$ is equivalent to minimizing $f(x)^2$, or $\|f(x)\|_2^2 = \sum_j f_j(x)^2$ in the multi-dimensional case. Conversely, global optima of differentiable functions are critical points of the function, i.e., solve $\nabla f(x) = 0$. So, there is an intimate connection between finding roots and optimization.

5.1 Univariate optimization via golden search

The textbook is not correct here. It argues that if f is continuous and has a global minimum m in $[a, b]$ and $a \leq x_l \leq x_r \leq b$ with $f(x_l) > f(x_l)$ then $m \geq x_l$. This is correct if f is convex: if $m \leq x_l$ then

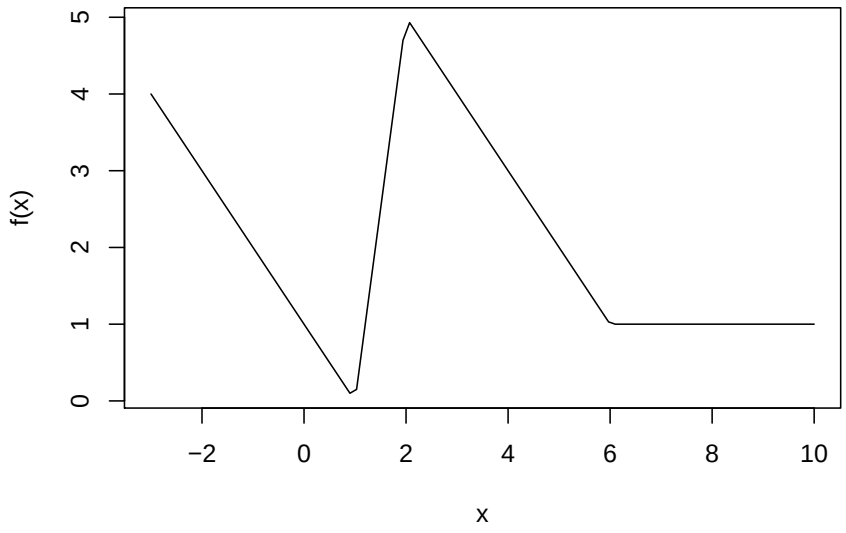
$$f(x_l) = f(\lambda m + (1 - \lambda)x_r) \leq \lambda f(m) + (1 - \lambda) f(x_r) \leq f(x_r)$$

which is impossible, but not in general. Consider

```
R> f <- function(x) {
+     ((1 - x) * (x < 1)
+      + 5 * (x - 1) * (x >= 1) * (x < 2)
+      + (7 - x) * (x >= 2) * (x < 6)
+      + (x >= 6))
+ }
```

which a global minimum at $x = 1$:

```
R> curve(f, -3, 10)
```



Using the code from the book:

```
R> golden <- function(f, a, b, tol = 1e-6) {
+     ratio <- 2 / (sqrt(5) + 1)
+     x1 <- b - ratio * (b - a)
+     x2 <- a + ratio * (b - a)
+     f1 <- f(x1)
+     f2 <- f(x2)
+     while(abs(b - a) > tol) {
+         if(f2 > f1) {
+             b <- x2
+             x2 <- x1
+             f2 <- f1
+             x1 <- b - ratio * (b - a)
+             f1 <- f(x1)
+         } else {
+             a <- x1
+             x1 <- x2
+             f1 <- f2
+             x2 <- a + ratio * (b - a)
+             f2 <- f(x2)
+         }
+     }
+     (a + b) / 2
+ }
```

we obtain

```
R> golden(f, 0, 7)

[1] 7
```

which is wrong.

What the golden section search really does is construct a sequence of bisections $a < x_m < b$ for which either eventually $f(x_m) \leq \min(f(a), f(b))$ or $x_m \to \{a, b\}$. This does not necessarily converge to a local minimum, though. If it does and the function is such that every local minimum is a global one (e.g., if convex), then a global minimum was found.

Note also that in each step the interval length reduces from $b - a$ to

$$(1 - 1/\phi)(b - a), \qquad 1 - 1/\phi = 1 - 2/(1 + \sqrt{5}) = \frac{\sqrt{5} - 1}{\sqrt{5} + 1} < 1$$

so decays to zero exponentially fast, and that in can be shown that using the golden ratio gives the optimal (in terms of shrinkage) bracketing scheme.

5.2 Univariate optimization via Newton's method

This is really finding the roots of f'.

5.3 Univariate optimization via `optimize`

Method uses a combination of golden section search and successive parabolic interpolation, and was designed for use with continuous functions.

If computed points are always unimodal then `optimize()` approximates a local minimum.

Example: MLE for Poisson distribution. The density of the Poisson distribution with parameter λ at (non-negative integer) x is

$$\frac{\lambda^x}{x!} e^{-\lambda}$$

so the likelihood of a sample $x_1, \ldots, x_n$ is

$$L(\lambda | x_1, \ldots, x_n) = \prod_{i=1}^{n} \frac{\lambda^{x_i}}{x_i!} e^{-\lambda}$$

and up to an additive constant, the log-likelihood is

$$LL(\lambda | x_1, \ldots, x_n) = \sum_{i=1}^{n} (x_i \log(\lambda) - \lambda) = s(x_1, \ldots, x_n) \log(\lambda) - n\lambda$$

with $s(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$.

The MLE is obviously given by $\hat{\lambda} = s/n = \bar{x}$.

Numerically:

```
R> mle_pois <- function(x) {
+     LL <- function(lambda) {
+         sum(x) * log(lambda) - length(x) * lambda
+     }
+     optimize(LL, lower = 0, upper = max(x), maximum = TRUE)
+ }
```

(Some finite upper bound is needed.)

To illustrate:

```
R> x <- rpois(100, 3.2)
R> mle_pois(x)

$maximum
[1] 3.469999

$objective
[1] 84.72164

R> ## Compare to
R> mean(x)

[1] 3.47
```

5.4 Multivariate optimization

Newton-type methods (as employed by `nlm()` can be motivated as for the univariate case: if f is twice continuously differentiable, then

$$\nabla f(y) \approx \nabla f(x) + H(x)(y - x),$$

where ∇f and H are the gradient and Hessian of f, respectively. Approximately solving $\nabla f(y) = 0$ thus gives

$$y = x - H(x)^{-1} \nabla f(x).$$

Available methods for general-purpose multi-variate optimization are `nlm()` (Newton-type), `optim()` (Nelder-Mead, quasi-Newton and conjugate-gradient algorithms, with options for box-constrained optimization and simulated annealing), and `nlminb()`.

`constrOptim()` performs minimization under linear constraints using an adaptive barrier (interior point) algorithm, but needs a feasible starting value.

For linear and non-linear regression, there are better routines (`nls()` for the latter).

Example: MLE for normal distribution. The density of the normal distribution with parameters μ and σ^2 at x is given by

$$\frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{(x-\mu)^2}{2\sigma^2}\right)$$

so the likelihood of a sample $x_1, \ldots, x_n$ is

$$L(\mu, \sigma^2 | x_1, \ldots, x_n) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{(x_i-\mu)^2}{2\sigma^2}\right)$$

and up to an additive constant, the log-likelihood is

$$LL(\mu, \sigma^2) = -\frac{n}{2} \log(\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (x_i - \mu)^2.$$

The MLEs are easily obtained as

$$\hat{\mu} = \bar{x}, \qquad \hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

(note that the MLE of σ^2 is not the sample variance!).

Numerically (minimizing twice the negative log-likelihood):

```
R> mle_norm <- function(x, p0) {
+     nLL <- function(p) {
+         mu <- p[1]
+         sigmasq <- p[2]
+         length(x) * log(sigmasq) + sum((x - mu) ^ 2) / sigmasq
+     }
+     optim(p0, nLL)
+ }
```

To illustrate (note that in R the normal distribution is parametrized by the standard deviation σ and not the variance):

```
R> x <- rnorm(100, 0.5, 2)
R> mle_norm(x, c(0, 1))

$par
[1] 0.3234268 4.2761782

$value
[1] 245.293

$counts
function gradient
      57       NA

$convergence
[1] 0

$message
NULL

R> ## Compare to
R> c(mean(x), sqrt(1 - 1 / length(x)) * sd(x))

[1] 0.3237381 2.0677581
```

(and note again that sd is not the MLE of σ).

Equivalently, using `nlm()` which employs a Newton-type method (and always minimizes), without explicitly providing gradients and hessians, and using the '`...`' argument instead of writing an inner function for computing the log-likelihood with starting value $(0, 1)$:

```
R> nLL2 <- function(p, x) {
+     mu <- p[1]
```

```
+     sigmasq <- p[2]
+     length(x) * log(sigmasq) + sum((x - mu) ^ 2) / sigmasq
+ }
R> nlm(nLL2, c(0, 1), x)

$minimum
[1] 245.293

$estimate
[1] 0.323738 4.275628

$gradient
[1] 1.924150e-05 3.777039e-05

$code
[1] 1

$iterations
[1] 14
```

5.5 Linear programming

In addition to package lpSolve, there are packages Rglpk, Rsymphony and Rcplex providing interfaces to the GLPK, COIN-OR SYMPHONY and CPLEX (commercial) solvers.

We illustrate some of the examples in the book using Rglpk.

Example 7.3 on page 143. We need to solve

$$\min C = 5x_1 + 8x_2$$

subject to the constraints

$$x_1 + x_2 \geq 2, \qquad x_1 + 2x_2 \geq 3$$

and

$$x_1, x_2 \geq 0.$$

Using Rgplk:

```
R> require("Rglpk")
R> obj <- c(5, 8)
R> mat <- rbind(c(1, 1), c(1, 2))
R> dir <- c(">=", ">=")
R> rhs <- c(2, 3)
R> Rglpk_solve_LP(obj, mat, dir, rhs)

$optimum
[1] 13

$solution
```

```
[1] 1 1

$status
[1] 0

$solution_dual
[1] 0 0

$auxiliary
$auxiliary$primal
[1] 2 3

$auxiliary$dual
[1] 2 3


$sensitivity_report
[1] NA
```

Example 7.5 on page 145. We need to solve

$$\max C = 5x_1 + 8x_2$$

subject to the constraints

$$x_1 + x_2 \leq 2, \qquad x_1 + 2x_2 = 3$$

and

$$x_1, x_2 \geq 0.$$

Using Rglpk:

```
R> obj <- c(5, 8)
R> mat <- rbind(c(1, 1), c(1, 2))
R> dir <- c("<=", "==")
R> rhs <- c(2, 3)
R> Rglpk_solve_LP(obj, mat, dir, rhs, max = TRUE)


$optimum
[1] 13

$solution
[1] 1 1

$status
[1] 0

$solution_dual
[1] 0 0

$auxiliary
```

```
$auxiliary$primal
[1] 2 3

$auxiliary$dual
[1] 2 3


$sensitivity_report
[1] NA
```

Example 7.11 on page 150. We need to solve

$$\min C = 2x_1 + 3x_2 + 4x_3 - x_4$$

subject to the constraints

$$x_1 + 2x_2 \geq 9, \qquad 3x_2 + x_3 \geq 9, \qquad x_2 + x_4 \leq 10$$

and

$$x_2, x_4 \quad \text{integer.}$$

Using Rglpk:

```
R> obj <- c(2, 3, 4, -1)
R> mat <- rbind(c(1, 2, 0, 0), c(0, 3, 1, 0), c(0, 1, 0, 1))
R> dir <- c(">=", ">=", "<=")
R> rhs <- c(9, 9, 10)
R> types <- c("C", "I", "C", "I")
R> Rglpk_solve_LP(obj, mat, dir, rhs, types)

$optimum
[1] 8

$solution
[1] 0.0 4.5 0.0 5.5

$status
[1] 0

$solution_dual
[1] 0 0 4 0

$auxiliary
$auxiliary$primal
[1]  9.0 13.5 10.0

$auxiliary$dual
[1]  2  0 -1


$sensitivity_report
[1] NA
```

5.6 Quadratic programming

Package quadprog solves quadratic programming problems of the form

$$-c'x + x'Dx/2 \to \min$$

subject to the constraints

$$A'x \geq b.$$

The portfolio selection problem in the textbook is written in the form

$$m'x - \frac{k}{2}x'Vx \to \max$$

subject to the constraints

$$\sum_{i=1}^{n} x_i = 1, \qquad x \geq 0,$$

i.e., $c \leftrightarrow m$ and $D \leftrightarrow kV$ (and not $kV/2$ as implied by example 7.14 which has $k = 4$ and uses $2V$), and $A \leftrightarrow [e, I_n]$.

Revisiting Example 7.14:

```
R> require("quadprog")
R> D <- matrix(0.002, 3, 3)
R> diag(D) <- 0.010
R> m <- c(0.002, 0.005, 0.010)
R> A <- cbind(rep(1, 3), diag(nrow = 3))
R> b <- c(1, rep(0, 3))
R> ## Note that 'meq' specifies the number of leading equality
R> ## constraints.
R> solve.QP(4 * D, m, A, b, meq = 1)

$solution
[1] 0.21875 0.31250 0.46875

$value
[1] 0.00315625

$unconstrained.solution
[1] -0.01339286  0.08035714  0.23660714

$iterations
[1] 2 0

$Lagrangian
[1] 0.013 0.000 0.000 0.000

$iact
[1] 1
```

To reproduce the results of Example 7.14:

```
R> solve.QP(2 * D, m, A, b, meq = 1)

$solution
[1] 0.1041667 0.2916667 0.6041667

$value
[1] -0.002020833

$unconstrained.solution
[1] -0.02678571  0.16071429  0.47321429

$iterations
[1] 2 0

$Lagrangian
[1] 0.003666667 0.000000000 0.000000000 0.000000000

$iact
[1] 1
```