Statistical Computing 1

Anthony Stephenson

Contents

Reproducibility and literate programming

Reproducibility

The general concept of reproducibility in scientific computing is the idea that any experiment and any results of experiments ought to be reproducible at a later date, by another person, on a different computer. In other words, if a researcher runs some computational experiment and gets some results, they should document the entire process they used, from their software environment and package versions upto any hyperparameter values and fitting procedures.

A non exhaustive list of relevant considerations are the following:

- Hardware used
- Software used, including language version, package versions, dependencies and environment (e.g. OS)
- Raw data source
- Any and all processes applied to the data for e.g. cleaning/augmentation and appropriate software packages used etc (as above)
- Model and fitting algorithms used, with all tunable parameters specified, or with their own fitting algorithms specified.
- Random seed used for any non-deterministic algorithms.
- Readability of code used

In short, every choice a researcher can plausibly make during the steps of their research should be documented clearly so that they (and any subsequent person) can carry out *exactly* what was done originally. For small datasets especially, it's easy to check that arbitrary choices can have significant effects on optimised parameters and/or performance, for a suitable measure.

Literate programming

In a similar vein to the ideas of reproducibility, literate programming is the name given to the idea that a program should be straightforwardly readable by a new party that has not been involved in its production. In particular, it emphasises the use of integrated documentation of code with explanations of what the code does. The idea is that it reads as a coherent text, interspersed with snippets of relevant code that exemplify the concept at that point in the text. This contrasts with other documentation and readability practices that stress interpretable variable names, clear commenting and separate documentation of functions and so on. The fundamental difference being that in literate programming the main body is the text, augmented with appropriate code, as opposed to the main body being the code, augmented with comments and supplemented with additional documentation. In practice, notebooks (e.g. Jupyter) or markdown is generally used to generate such documents.

As an example, I will solve a Project Euler problem in the form of a notebook.

Project Euler problems

Largest prime factor

The prime factors of 13195 are 5, 7, 13 and 29.

What is the largest prime factor of the number 600851475143 ?

In order to solve this problem, we'll need to define some useful functions first.

In particular, some sort of prime number finding algorithm, for which (after a quick google search) we'll use the Sieve of Eratosthenes since that comes up immediately:

```r
# An algorithm used to determine prime numbers upto some number n. Function takes
# the scalar n and returns an array of primes. Pseudo-code taken from wikipedia.
sieve_of_eratosthenes <- function(n) {
    A <- rep(TRUE, n - 1)
    for (i in 2:floor(sqrt(n))) {
        if (A[i - 1]) {
            j <- i^2
            while (j < n+1) {
                A[j - 1] <- FALSE
                j <- j + i
            }
        }
    }
    candidates <- 2:n
    return(candidates[A])
}
```

We then need a function to tell us if a divisor is a factor of our target number, i.e. to determine that the output is an integer:

```
# Simple function using modulo to determine if an input variable x is an integer
# (in value, not type)
is_integer <- function(x) {
    return(x %% 1 == 0)
}
```

Which just makes use of the modulo operator.

Finally we define the function that actually checks the target variable for prime factors, by testing each of a set of candidate primes and recording each one that is a factor of the target. That prime is then removed from the list. This continues until t == 1 i.e. all have been found. If we run out of candidate primes before we find all of the prime factors, we calculate another set of candidate primes. (Inefficiently, as the algorithm starts at 2 each time.)

```
# An algorithm to find the prime factors of some input integer. This makes use
# of the Sieve of Eratosthenes algorithm to find primes upto some input n.
find_prime_factors <- function(target) {
    initial_guess <- 30
    test_primes <- sieve_of_eratosthenes(initial_guess)
    counter <- 0
    while (t > 2 & length(test_primes) > 0) {
        counter <- counter + 1
        for (p in test_primes) {
            while (is_integer(t / p)) {
                prime_factors <- c(prime_factors, p)
                t <- t / p
            }
            test_primes[-which(test_primes == p)]
        }
        if (t > 2) {
            test_primes <- sieve_of_eratosthenes(initial_guess + counter * 10)
        }
    }
    return(prime_factors)
}
```

Hence, the result for our test variable:

```
t <- 600851475143
prime_factors <- NULL
largest_prime <- max(find_prime_factors(t))
print(largest_prime)
```

```
## [1] 6857
```

Common R

Vectorisation

Vectorisation is the idea that in some interpreted languages (e.g. R and MATLAB) it is usually faster to use vectorised code rather than directly implementing for-loops to do the same task. The reason for this is there is an overhead associated with the interpretation of the R code, whereas in vectorised code, there is will be a function that carries out an equivalent loop behind the scenes, but in pre-compiled code.

To demonstrate this, we'll time how long it takes to run a couple of equivalent (in terms of functionality) snippets of code.

```
inner_product1 <- function(a, b) {
  n <- length(a)
  output <- 0
  for (i in 1:n) {
    output <- output + a[i] * b[i]
  }
}

inner_product2 <- function(a, b) t(a) %*% b

a <- log(1:1e7)
b <- log(1:1e7)
system.time(inner_product1(a, b))
```

```
##    user  system elapsed
##   0.339   0.000   0.340
```

```
system.time(inner_product2(a, b))
```

```
##    user  system elapsed
##   0.057   0.001   0.058
```

```
remove(a, b)
```

Clearly the results from the second function, which utilises R matrix operations, is much faster.

R apply family

The "apply" functions effectively call a function repeatedly for each element or dimension of some generalised input array.

For example, `apply` acts on some dimension of an array, specified by `MARGIN`, e.g. by squaring each element:

```
x <- c(1, 3, 5, 6, 7)
y <- apply(t(x), 1, function(a) a^2)
```

Note that we have to use `t(x)` to ensure that the input is an array with a non-`NULL` dimension.

The related function `lapply` meanwhile applies a function to each element of an array or list, returning a new list.

```
x <- list(c(1, 2), c(2, 4), c(5, 6))
y <- lapply(x, mean)
print(y)
```

```
## [[1]]
## [1] 1.5
##
## [[2]]
## [1] 3
##
## [[3]]
## [1] 5.5
```

Then we have `sapply` which does the same thing as either of the other two, but returns a "simplified" output, e.g. in the example snippets above, it returns a vector in the form of the original vector `x`.

Parallel programming

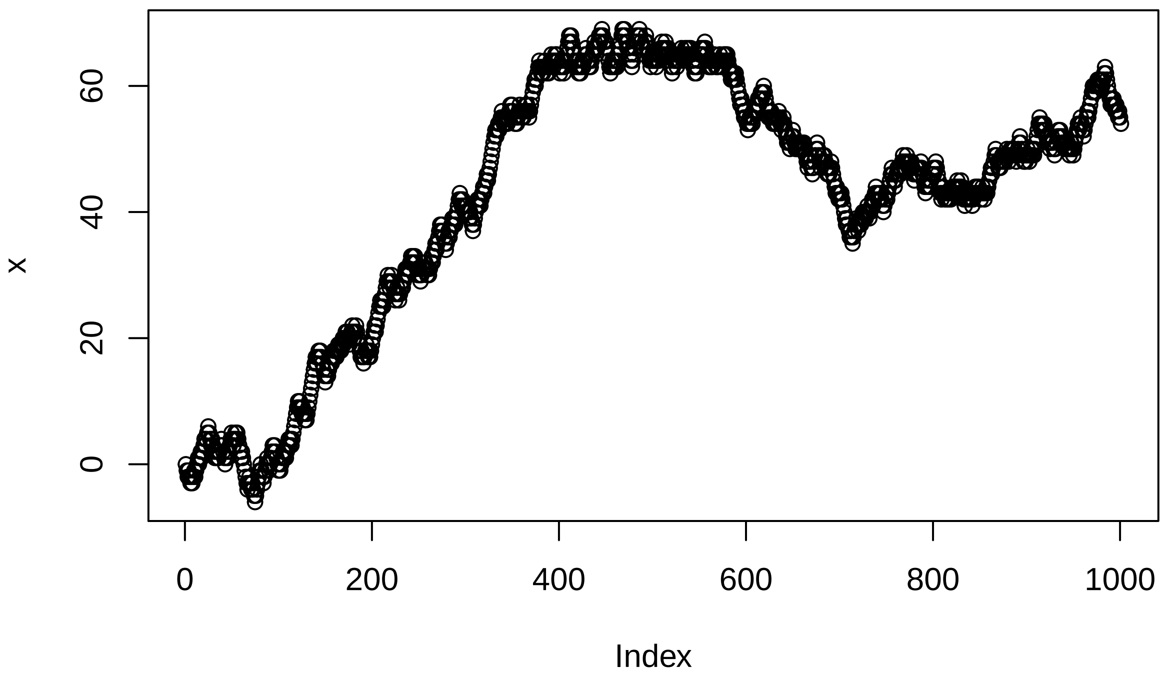
Parallel programming is designed to speed up CPU intensive tasks that run large quantities of independent computations (or at least some independent partitions of those tasks). Usually, one will specify a number of cores to use for a task and then utilise some form of parallel indicator in the code to divide the task amongst the cores designated. Note that there is some overhead associated with spinning up additional cores and for distributing the tasks. In particular, there can be significant memory costs to parallel programming if, say, you want to run some data processing on a large dataset a bunch of times as each core will need independent access to the original data set in order to run the processing.

In R we can run parallel programs using either `mclapply` or `foreach` with `doParallel`. Since I am running this on Windows I will skip the example with `mclapply` as it reportedly doesn't work.

As an example, we'll run 500 random walk simulations with `foreach`; first without parallelisation and then with.

```r
# generate points from a random walk
rw <- function(n){
  x <- rep(0, n)
  for (i in 1:n) {
    u <- runif(1)
    if (u < 0.5) {
      x[i+1] <- x[i] + 1
    }
    else {
      x[i+1] <- x[i] - 1
    }
  }
  return(x)
}

x <- rw(1e3)
plot(x)
```



Run the random walk `n_runs` times and time the duration of the calls with `system.time`.

```
library(foreach)

n_runs = 500
n_steps = 1000
system.time(foreach (i=1:n_runs, .combine=rbind, .final=function(x) x[,n_steps]) %do% {
  ts <- rw(n_steps)
  tdiff <- diff(ts)
})
```

```
##    user  system elapsed
##   0.980   0.097   1.079
```

Now repeat that, but using parallelisation.

```
library(parallel)
library(doParallel)
```

```
## Loading required package: iterators
```

```
cores <- detectCores()
cl <- makeCluster(cores[1]-1)
registerDoParallel(cl)

system.time(foreach (i=1:n_runs, .combine=rbind, .final=function(x) x[,n_steps]) %dopar% {
  ts <- rw(n_steps)
  tdiff <- diff(ts)
})
```

```
##    user  system elapsed
##   0.182   0.026   0.469
```

```
stopCluster(cl)
```

As we would hope, the parallel version runs significantly faster, at around half the time with 3 cores, so it scales worse than the number of cores.

Tidyverse

The "Tidyverse" is the name given to a collection of intercompatible R packages that share a common philosophy centred around the manipulation (and plotting) of `data.frames`. The general idea is that it is easier, more reproducible and less bug-prone than working with base R.

Some of the key packages included in the tidyverse are:

- `magrittr` - implements pipes. There are multiple different types of pipes:
  - `%>%` the standard pipe operator, which effectively implements function composition. i.e. `x %>% abs %>% log` is equivalent to `log(abs(x))`.
  - `%T>%` the "tee" operator, which returns the left hand side value.
  - `%$%` the "exposition" pipe operator which exposes the names of variables on the LHS to the function on the right, as in the following snippet

  ```
  data.frame(z = rnorm(100)) %$%
  ts.plot(z)
  ```

  - `%<>%` the compound assignment pipe operator is a shorthand that can be used for the first pipe in a sequence where a variable is assigned the output of the sequence, e.g. `foo <- foo %>% bar %>% baz` is equivalent to `foo %<>% bar %>% baz`.
- `tidyr` - functions for reshaping `data.frames` e.g. by converting between long and wide formats using `gather()` and `spread()` which essentially act as pivot functions.
- `dplyr` - functions for transforming `data.frames`, e.g. by selecting subsets of data, or filtering, or grouping data (`select()`, `filter()`, `group_by`, …)
- `ggplot2` - plotting functions. The base function `ggplot` takes a dataframe as it's argument and is then added to with supplementary functions that actually plot the data. E.g. `geom_point` plots a scatter plot. Usage is essentially something like `ggplot(df, x = col_of_df, y = other_col_of_df) + geom_point()`.

Example usage

To demonstrate some of their usage, we will use the Kaggle Ashrae Energy Prediction data, found at https://www.kaggle.com/c/ashrae-energy-prediction/overview.

First we'll load some of the downloaded data and take a look at it, see what we're dealing with.

```
data_path <- "/Users/anthonystephenson/Downloads/ashrae-energy-prediction"
building_metadata <- read.table(paste(data_path, "building_metadata.csv", sep="/"), sep=",", header=TRUI
head(building_metadata)
```

```
##   site_id building_id primary_use square_feet year_built floor_count
## 1       0           0   Education        7432       2008          NA
## 2       0           1   Education        2720       2004          NA
## 3       0           2   Education        5376       1991          NA
## 4       0           3   Education       23685       2002          NA
## 5       0           4   Education      116607       1975          NA
## 6       0           5   Education        8000       2000          NA
```

```
train <- read.table(paste(data_path, "train.csv", sep="/"), sep=",", header=TRUE)
head(train)
```

```
##   building_id meter           timestamp meter_reading
## 1           0     0 2016-01-01 00:00:00             0
## 2           1     0 2016-01-01 00:00:00             0
## 3           2     0 2016-01-01 00:00:00             0
## 4           3     0 2016-01-01 00:00:00             0
## 5           4     0 2016-01-01 00:00:00             0
## 6           5     0 2016-01-01 00:00:00             0
```

```r
weather_train <- read.table(paste(data_path, "weather_train.csv", sep="/"), sep=",", header=TRUE)
head(weather_train)
```

```
##   site_id           timestamp air_temperature cloud_coverage dew_temperature
## 1       0 2016-01-01 00:00:00            25.0              6            20.0
## 2       0 2016-01-01 01:00:00            24.4             NA            21.1
## 3       0 2016-01-01 02:00:00            22.8              2            21.1
## 4       0 2016-01-01 03:00:00            21.1              2            20.6
## 5       0 2016-01-01 04:00:00            20.0              2            20.0
## 6       0 2016-01-01 05:00:00            19.4             NA            19.4
##   precip_depth_1_hr sea_level_pressure wind_direction wind_speed
## 1                NA             1019.7              0        0.0
## 2                -1             1020.2             70        1.5
## 3                 0             1020.2              0        0.0
## 4                 0             1020.1              0        0.0
## 5                -1             1020.0            250        2.6
## 6                 0                 NA              0        0.0
```

```r
train %>% object.size %>% format("MB")
```

```
## [1] "463.4 Mb"
```

```r
weather_train %>% object.size %>% format("MB")
```

```
## [1] "9.7 Mb"
```

```r
building_metadata %>% object.size %>% format("MB")
```

```
## [1] "0 Mb"
```

Now we want to reformat some of the data. In particular, changing the timestamp column, currently a `<chr>` (i.e. char) type into a datetime format that can be interpreted by e.g. `ggplot`. For this, we'll use the tidyverse package `lubridate`.

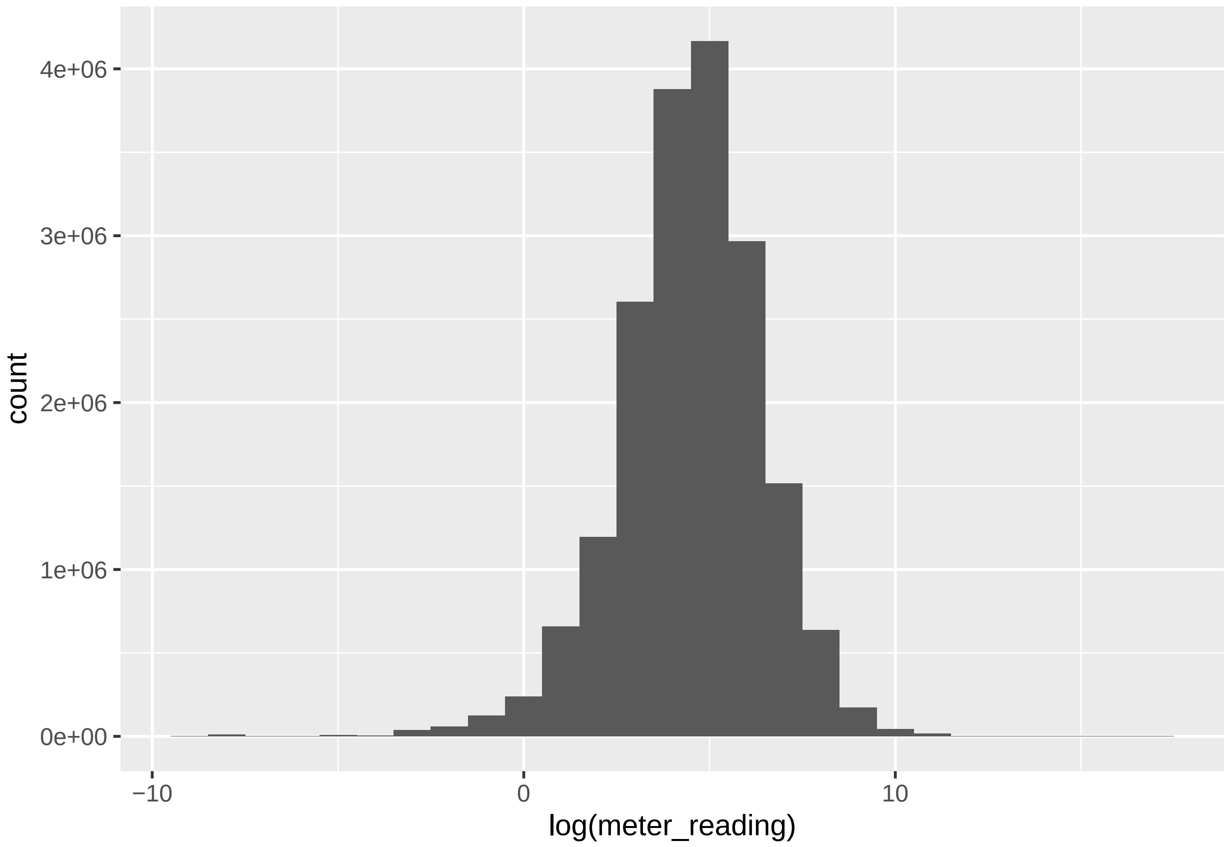
```r
train <- transform(train, timestamp = ymd_hms(timestamp, tz="GMT"))
weather_train <- transform(weather_train, timestamp = ymd_hms(timestamp, tz="GMT"))
head(train)
```

```
##   building_id meter  timestamp meter_reading
## 1           0     0 2016-01-01             0
## 2           1     0 2016-01-01             0
## 3           2     0 2016-01-01             0
## 4           3     0 2016-01-01             0
## 5           4     0 2016-01-01             0
## 6           5     0 2016-01-01             0
```

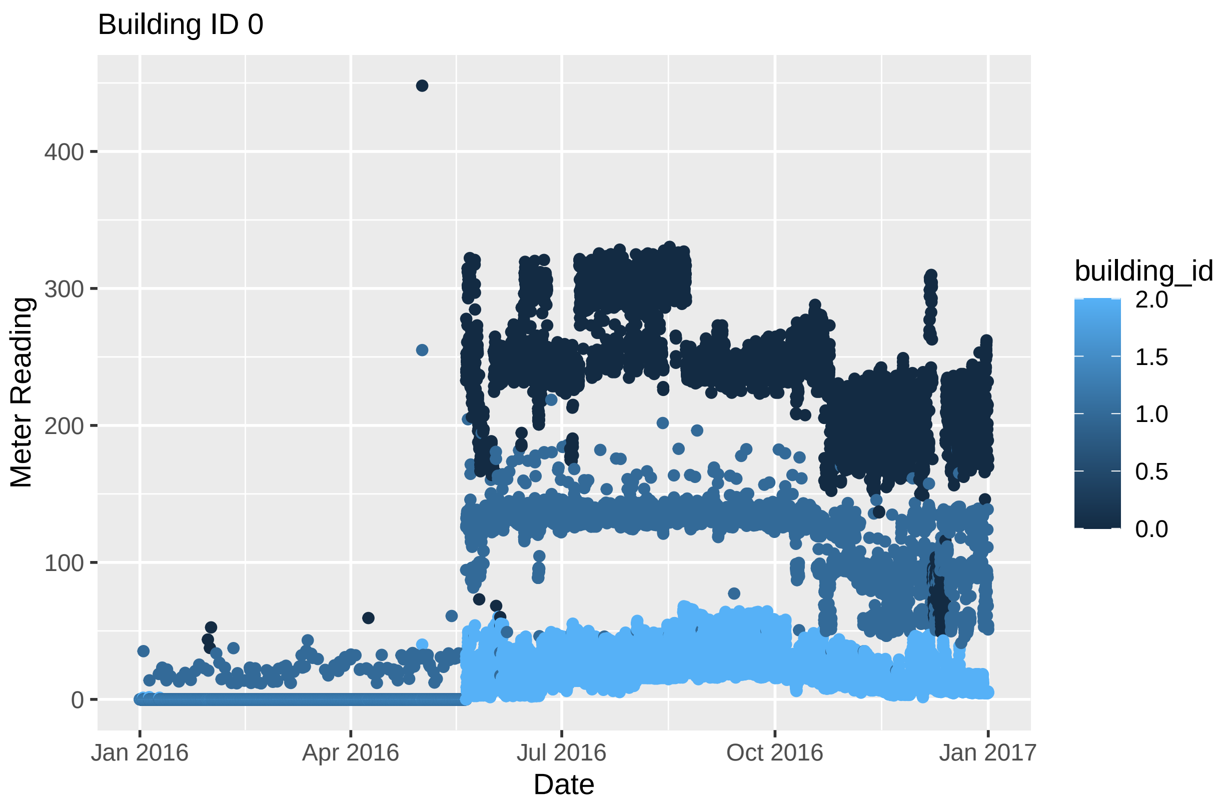
To get a feel for the data, we want to visualise some of it. Here we make use of the `%>%` pipe operator to simplify the readability of the expression that takes the dataset and manipulates it into a suitable form for our plotting purposes. For example, in the first plot we only want to plot the target variable, `meter_reading`, so we `select` that from the `train data.frame`. We then decide to skip rows where the meter_reading is 0, to get a feel for the kind of non-zero values we expect, and the distribution of them. Finally, we pass this modified subset of `train` to `ggplot` with a `geom_histogram` layer.

```
ids <- c(0, 1, 2)

# plot a histogram of the log of non-zero meter readings (as the data is sparse)
train %>%
  select(building_id, meter_reading) %>%
  filter(meter_reading != 0) %>%
  ggplot(aes(x = log(meter_reading))) + geom_histogram(binwidth = 1)
```



```
# plot a few example locations' meter readings against time
train %>%
  select(building_id, timestamp, meter_reading) %>%
  filter(building_id %in% ids) %>%
  ggplot(aes(x = timestamp, y = meter_reading, color = building_id)) + geom_point() + labs(title = "",
           subtitle = "Building ID 0",
           x = "Date", y = "Meter Reading")
```



Now we want to combine some of the datasets for further analysis, since we imagine that weather will impact energy usage, for instance. I am using `inner_join` here to only include rows where we have data from each source present.

```
buildings <- inner_join(train, building_metadata, by="building_id")
all_data <- inner_join(buildings, weather_train, by=c("site_id", "timestamp"))
head(all_data)
```

```
##   building_id meter  timestamp meter_reading site_id primary_use square_feet
## 1           0     0 2016-01-01             0       0   Education        7432
## 2           1     0 2016-01-01             0       0   Education        2720
## 3           2     0 2016-01-01             0       0   Education        5376
## 4           3     0 2016-01-01             0       0   Education       23685
## 5           4     0 2016-01-01             0       0   Education      116607
## 6           5     0 2016-01-01             0       0   Education        8000
##   year_built floor_count air_temperature cloud_coverage dew_temperature
## 1       2008          NA              25              6              20
## 2       2004          NA              25              6              20
## 3       1991          NA              25              6              20
## 4       2002          NA              25              6              20
## 5       1975          NA              25              6              20
## 6       2000          NA              25              6              20
##   precip_depth_1_hr sea_level_pressure wind_direction wind_speed
## 1                NA             1019.7              0          0
## 2                NA             1019.7              0          0
## 3                NA             1019.7              0          0
## 4                NA             1019.7              0          0
## 5                NA             1019.7              0          0
## 6                NA             1019.7              0          0
```

Note that doing this increases the overall size of the data.

```r
all_data %>% object.size %>% format("MB")
```

```
## [1] "1996.1 Mb"
```

Packages

Projects

Projects exist in effectively all languages as a way to organise code into sensible (mostly) self-contained sections, designed to do a particular task/solve a particular problem. In R they usually follow the structure:

```
project
 ├── R
 ├── README.md
 ├── data
 ├── doc
 └── output
```

with the actual R files in the `R` directory (without subdirectories) and any data in the `data` directory, potentially organised by separating raw and R-modified datasets. `doc` contains relevant documentation, such as source files for an associated paper and `output` containing results of analysis carried out by the code on the data provided. Note that this is a less formal structure than actual packages.

Version Control

Version control is a way of keeping track of changes made to files over the lifecycle of a project. There are a few different systems out there, but the most popular is comfortably `git`, often with a remote repository provided by a server such as *GitHub*. The idea is that for each change made to a file, the change is *committed* to the repository with a suitably explanatory comment so that future coders (possibly the same person, possibly not) can see and understand what was done and why. It also facilitates the possibility of "rolling back" a version of code, if a singlular commit or series of commits subsequently introduced a bug.

Going into a great deal of detail on the various functionalities of git is beyond the scope of this document, but a brief description of some of them follows.

- Stashing - Sometimes when we are in the middle of making changes to some files in order to solve a specific problem, but are not yet ready to commit them, a new more urgent problem emerges. To avoid discarding the work we have already done, we can `stash` the changes, clearing our current changes, but storing them for later. We can now work on the new problem and when finished, `pop` the stashed changes back (potentially merging with any changes introduced by the interrupting issue).
- Branches - Branches are a very useful way to organise parallel changes to a project (see *Git Workflow* for one methodology). For example, say we want to implement two different but potentially conflicting features in a project, they can each be developed in a separate branch to "main" and, when finished, merged into "main". This effectively allows asynchronous changes to be made to all branches, and allows one to test changes before merging to "main" to avoid introducing bugs into a live environment. (Ideally one would probably use multiple environments hosted independently if this is a real concern and use a *staging* environment to test changes.)
- Revert - As mentioned above, when later commits introduce bugs, one can revert changes to an older version, before the damage was done.
- History - One can view the history of a file and see all previous changes made to it, which can be a very useful way to track down bugs, and see when (say) a function was added or removed and by whom.
- Working trees - Working trees allow the duplication of a projects code into a new directory when a branch is created. This can be useful for comparing the output of a new feature implementation

compared to an old one, for example.

Packages

Packages exist in most languages and are a more formal way to organise and share code. They should be self-contained and generate reproducible results. In addition to the files and folders typically found in a project, as described above, a package will usually contain a licence that describes how the code may be used; a namespace that describes what functions the package will make available to others and what dependencies on other packages it has; R documentation for the code in the `man` directory and unit tests for the functions in the `tests` directory.

Package structure:

```
project
 ├── DESCRIPTION
 ├── LICENSE
 ├── LICENSE.md
 ├── NAMESPACE
 ├── R
 ├── man
 └── tests
```

Unit testing

Unit testing is an important tool for writing reliable, reproducible code. The general idea is that for each function in our package, there should be an associated test that checks whether the function correctly does it's job, i.e. for a given test input, does it produce the expected output? In practice, this can be difficult to do infallibly - edge cases can sometimes be tricky to think of a priori (before they inevitably emerge in use) and writing comprehensive tests for functions can be a little time consuming. It tends to be very helpful in the long run for complicated functions and packages, however, especially as function internals are changed over time and we want to check that the function still behaves as originally intended. In R, the go-to package is `testthat` with the following naming convention for test files: `test-<name>`. An toy test might be:

```
source("~/Documents/GitHub/compass-homeworks/R/linmodel/tests/testthat/test-linmodel.R")
test_that("multiplication works", {
  expect_equal(2 * 2, 4)
})
```

At some point we want to check that a) we have good *coverage* of our code, i.e. there are tests for all/most of our functions and b) our code actually passes the tests. To do these things we can use `covr` and `devtools` as in the snippet below.

```
setwd("~/Documents/GitHub/compass-homeworks/R/linmodel/R")
devtools::test()
```

```
## Loading linmodel

## Testing linmodel

## v |  OK F W S | Context
## / |   0       | linmodelv |   1       | linmodel
##
## == Results ===================================================================
## OK:       1
## Failed:   0
## Warnings: 0
## Skipped:  0
```

```
covr::report()
```

Continuous Integration

Continuous integration (CI) is a useful tool for automating some of the infrastructure around developing a package and using source control. For example, when pushing to a remote server we might want to run tests to ensure the code passes before it can be run on the server. We might also want to trigger tests for different operating systems or versions. As usual, there are multiple options for managing CI, including *Travis* and *Jenkins*.

Object Oriented Programming

Object oriented programming (OOP) is defined by the implementation of classes with the following properties:

- Polymorphism - this term describes the idea that objects of different types can be accessed through the same interface. In practice this means that a particular class can implement a method that returns some object or value that we understand from the name of this method. If we now take a different class that implements a method of the same name, we expect it to return a value or object that relates directly to the output of the first class. The actual details of the implementation of the two classes can be entirely different, so long as the behaviour is equivalent.
- Inheritance - technically not required, but usually classes can be defined in a hierarchical manner such that *child* classes *inherit* unimplemented methods from their `parent` class. This can be useful for programming objects with hierarchical levels of specificity. Any method that exists in a parent but is reimplemented in the child is considered to be *overloaded*.

There are a few different packages that implement these properties; S3, S4 and Reference Classes. S3 and S4 have been effectively superseded by the Reference Classes package so here we will primarily describe the functionality of just that implementation, after a brief introduction of the former two.

S3

If the user only wishes to implement a simple class which makes use of pre-existing generic R functions (e.g. `print()` and `plot()`) then S3 enables a quick and simple implementation. This is because S3 works by defining *methods* as normal R functions, but with the specific naming convention `[method_name].[class_name]()`, so all we need to do is define a function that acts on the kind of object we want with that naming convention, e.g. `print.foo <- function(x) print("foo")`. We can also define new generic functions in the following way:

```
 mean <- function (x, ...) {
   UseMethod("mean", x)
 }
```

and hence methods:

```
mean.numeric <- function(x, ...) sum(x) / length(x)
mean.data.frame <- function(x, ...) sapply(x, mean, ...)
mean.matrix <- function(x, ...) apply(x, 2, mean)
```

These examples are taken from the Advanced R website which contains a more detailed explanation of the functionality.

The implication though, is that the generic function name is a sort of template and we can have a set of functions that have type-dependent mechanics but intuitive functionality.

Of course we still need to define class instances, which we do either at initialization with

```
foo <- structure(list(), class = "foo")
```

or after creation, with

```
foo <- list()
class(foo) <- "foo"
```

Note that inheritance is implemented for S3 in the form of the `NextMethod` function, which is described on Advanced R.

S4

S4 is an extension to S3 based on the same principles but with a more formalised structure and syntax, leaning towards a more normal OOP construction.

We can define class instances with S4 using `setClass`, with associated methods (`setMethod`) and generic functions (`setGeneric`). As an example, we'll define a toy class that implements a complex number type with a print method, fields (`slots`) relating to its real and imaginary parts, and a parent class that it inherits from, listed in the `contains` argument.

```
setClass("Complex",
  contains = "numeric",
  slots = list(real = "numeric", imag = "numeric"))

setGeneric("conjugate", function(x) standardGeneric("conjugate"))
```

```
## [1] "conjugate"
```

```
# define a convenient print function
setMethod("print", "Complex",
  function(x, ...) {
    infix_symb <- ifelse(sign(x@imag) > 0, "+", "-" )
    cat(x@real, infix_symb, abs(x@imag), "i", "\n")
  }
)

# define the complex conjugate
setMethod("conjugate", "Complex",
  function(x){
    x@imag <- -x@imag
    return(x)
  }
)

# test the functionality by creating a new Complex object and using print and
# conjugate
z <- new("Complex", real = 1, imag =  2)
print(z)
```

```
## 1 + 2 i
```

```
z_dagger <- conjugate(z)
print(z_dagger)
```

```
## 1 - 2 i
```

Reference Classes

Reference Classes form a more standardized approach to OOP and contain similar structures to what would be found in other languages.

Fundamental difference between Reference Classes and S3/4:

- Methods belong to objects, rather than *generic functions*.
- Modify in-place semantics. Usually in R, objects are *passed by value* rather than *passed by reference* meaning that the data is copied whenever a variable is assigned or passed as an argument to a function. In Reference Classes, *pass by reference* is used meaning that we can modify a variable in-place which can lead to unintended side effects.

To do this, we will implement a simple linear regression model with a suitable class. First we will generate some data to use, before defining our class.

```r
n <- 10
X <- t(t(runif(n))) * 2
y <- exp(1.5 * X - 1) + rnorm(n, 0, 1)
```

In our class we define a set of `fields` followed by a set of methods, including an initialization routine. The declaration of methods here is more similar to other languages than S3 or S4. Also note that the object instance is referred to as `.self` vs `.Object` in S4.

```r
library(methods)
simple_lin_regression <- setRefClass("simple_lin_regression",
                             fields=c(response="matrix", regressor="matrix", estimate="numeric"))

simple_lin_regression$methods(
  initialize = function(response, regressor) {
    .self$response <- response
    .self$regressor <- regressor

    # define design matrix
    n <- length(response)
    D <- matrix(c(rep(1, n), regressor), ncol = 2)

    # compute the OLS estimate
    b <- solve(t(D) %*% D) %*% t(D) %*% response
    .self$estimate <- as.numeric(b)
  },
  # plot our raw data, x vs y
  plot_data = function() {
    plot(x = .self$regressor, y = .self$response,
         xlab = expression(X), ylab = expression(Y))
    abline(a = .self$estimate[1], b = .self$estimate[2])
  },
  # when print(object) is called, print out the top 10 rows of X and y, as well
  # as the model output and coefficients
  show = function() {
    cat("head(x) =", head(.self$regressor), "\n")
    cat("head(y) =", head(.self$response), "\n\n")
    cat("Estimated regression: E[y|x] = b0 + b1 * x\n")
    cat("b0 = ", .self$estimate[1], " and b1 = ", .self$estimate[2], ".\n", sep = "")
  },
  # Compute predictions on the objects data and then plot the residuals
  residual_analysis = function() {
    predictions <- .self$estimate[1] + .self$estimate[2] * .self$regressor
    plot(x=predictions, y=.self$response - predictions,
         xlab=expression(hat(Y)), ylab=expression(Y - hat(Y)))
    abline(0, 0)
  },
  # replace data in the object, and refit the model
  setData = function(y, X) {
    .self$initialize(response = y, regressor = X)
  }
)
```

Now to demonstrate the functionality, we will call the methods defined for the linear model class and build, test and plot the model attributes.

First build the object (and fit the model at the same time):

```r
slml <- simple_lin_regression$new(response=y, regressor=X)
```

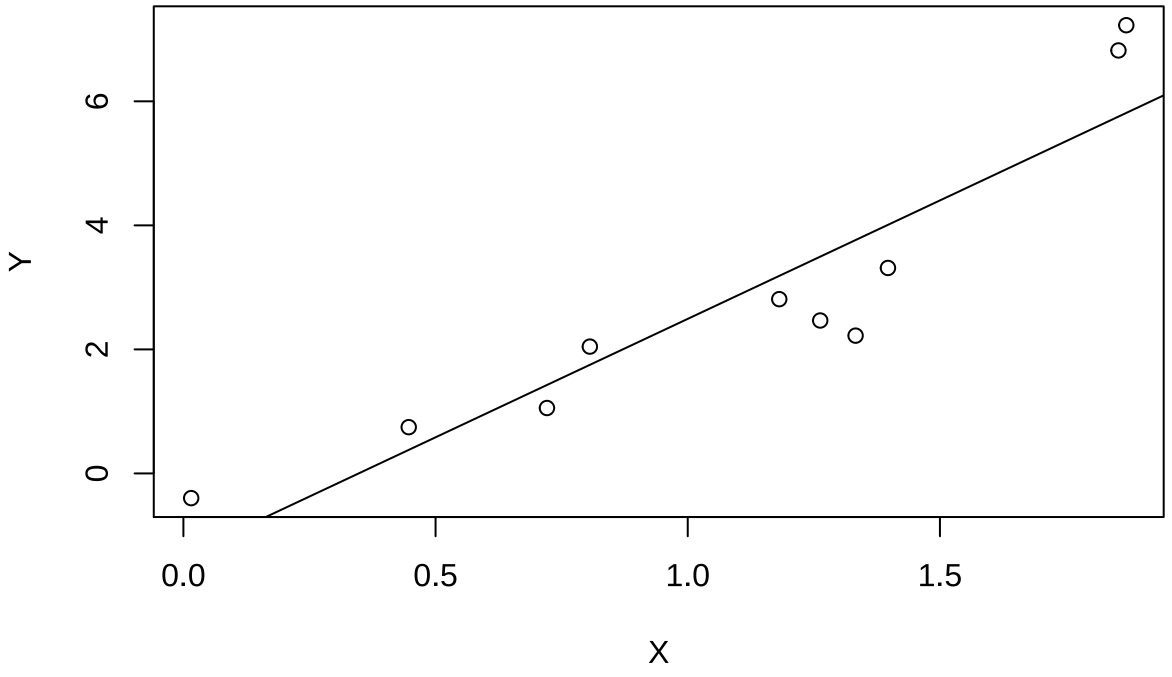
Now we call `print()` which calls the `show()` method we implemented:

```r
print(slml)
```

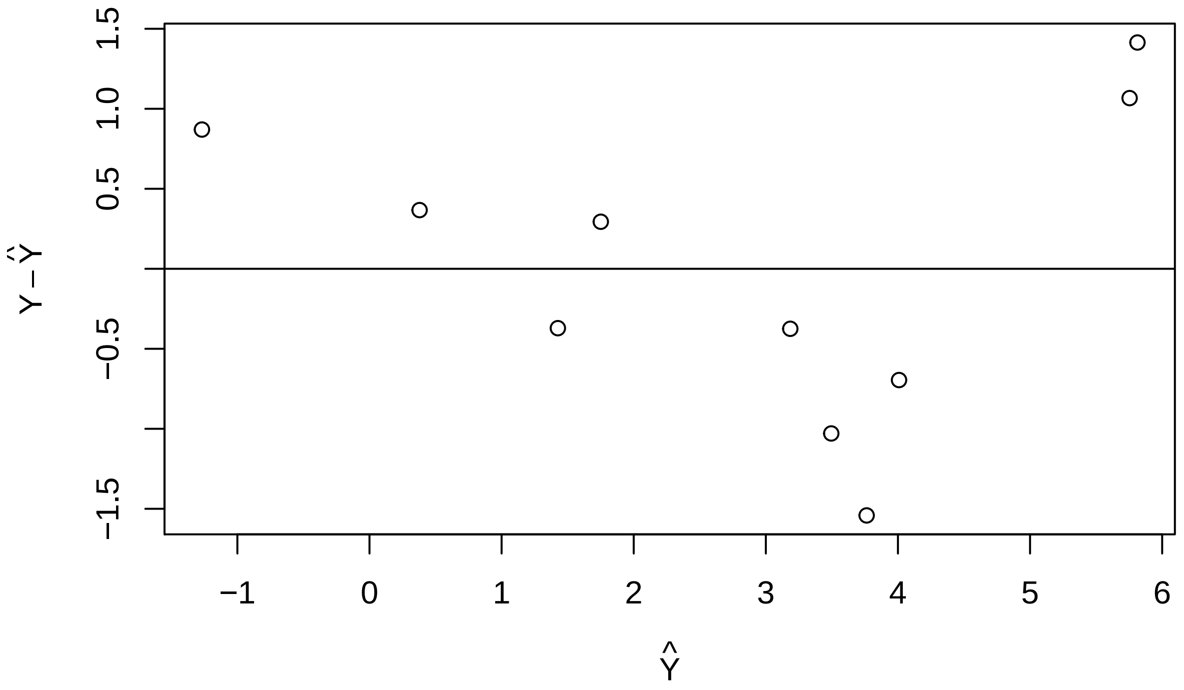
```
## head(x) = 1.396988 0.8059355 0.4467817 1.332814 1.181449 1.869519
## head(y) = 3.312956 2.044827 0.7459777 2.221839 2.810095 7.22719
##
## Estimated regression: E[y|x] = b0 + b1 * x
## b0 = -1.327282 and b1 = 3.819356.
```

Finally, we call our plot method to visualise the original data, and `residual_analysis()` to measure and visualise the residuals from out fit.

```r
slml$plot_data()
```



```r
slml$residual_analysis()
```



Functional Programming

In contrast to OOP, functional programming is based around the concept that complex programs can be built purely from functions, so long as the functions can have certain properties, which will be detailed in the following subsections. In practice almost no languages are actually purely functional, but rely on objects to some degree.

First class functions

First class functions exist in a given language if in the language one can pass functions as arguments, functions can also be returned by other functions and functions can be stored in data structures. These characteristics allow functions to behave as variables, and also enables the same sort of functionality as permitted by OOP to be reproduced, albeit under a different framework.

For example, say I want to build a kernel regression model, but I want to tune the hyper-parameters of the kernel, I can define a function for the kernel, that takes the hyperparameter as an argument, as a generic definition, but after tuning, I can define a *function factory* that takes this kernel function and a value for a hyperparameter as arguments and returns a partially called version of the kernel function, with the hyperparameter assigned:

```
# defines a polynomial kernel with order (hyper)parameter b
poly_kernel <- function(x_i, x_j, b) {
    return((t(x_i) %*% x_j + 1)^b)
}

# function factory that returns a partially called kernel that now takes only
# input arguments in locations x_i, i.e. k(x,x', b) -> k(x,x'; b)
tuned_kernel <- function(ckernel, ...) {
    function(x_i, x_j){
        ckernel(x_i, x_j, ...)
    }
}
```

Pure functions

Pure functions are functions that have no side effects, i.e. do not modify the global program state in any way. Effectively they behave like mathematical functions and as such their output should be straightforward to predict (and tests should be straightforward to construct).

Closures

Closures are related to scope. When a variable is declared, the environment in which it is declared determines its scope and hence its accessibility. For example, if a variable is declared within a loop or a function, it has a local scope which means that it only exists within the loop or function in which it was defined, so accessing it outside of that environment will fail. Sub-environments can access variables in their parent scope, however, so a nested function can access variables in its parent function's scope.

```
a <- 1
foo <- function(){
  b <- 2
  print(a)
}
foo()
```

```
## [1] 1
```

```
cat("a =", a, "\n")
```

```
## a = 1
```

```
print(b)
```

```
## Error in h(simpleError(msg, call)): error in evaluating the argument 'x' in selecting a method for fu
```

With closures, a function is declared in a given environment, with a corresponding scope, only this scope can persist beyond the original scope. In particular, if a function is declared in the scope of an existing function, any unresolved variables can be assigned by using their definitions in the parent scope. If the nested function is subsequently called *outside* of the original environment, these definitions will be retained, i.e. the scope will persist. Using the kernel tuner function factory we already defined, we can generate a new function `tuned_poly` with a persistent `b` parameter

```
tuned_poly <- tuned_kernel(poly_kernel, 1)
```

Lazy evaluation

Lazy evaluation describes the behaviour that an expression is not evaluated until required. This can save computational resource by not using as much memory or CPU as strict evaluation since until necessary, values will not be generated. Lazy evaluation can lead to confusing unexpected behaviour, on the other hand, and can cause memory leakage. We will use an example from Advanced R to demonstrate this, since it is very clear. The first example shows that the argument `exponent` is only evaluated when the function `square()` is called, rather than when the function is declared.

```
make.power.function <- function(exponent) {
  power <- function(x) x^exponent
}

exponent <- 2
square <- make.power.function(exponent)
exponent <- 3
square(2)
```

```
## [1] 8
```

In comparison, here we use the `force()` function and thus the `exponent` argument is evaluated at declaration and the subsequent assignment to `exponent` is now not associated with the calculation on the final line.

```r
make.power.function <- function(exponent) {
  force(exponent)
  power <- function(x) x^exponent
}
exponent <- 2
square <- make.power.function(exponent)
exponent <- 3
square(2)
```

```
## [1] 4
```

Note that since R 3.2.0 "higher order functions such as the `apply` functions and `Reduce()` now force arguments to the functions they apply in order to eliminate undesirable interactions between lazy evaluation and variable capture in closures." (https://cran.r-project.org/src/base/NEWS).