Statistical Programming

Michaelmas Term, 2017

Marco Scutari

scutari@stats.ox.ac.uk
Department of Statistics
University of Oxford

January 9, 2018

Course Information

Lectures

Week 1:	Monday/Wednesday 10am, Tuesday/Thursday 10am (Worksheets)
Week 2:	Monday/Wednesday 10am, Tuesday/Thursday 10am (Worksheets)
Week 3:	Monday/Wednesday 10am, Tuesday/Thursday 10am (Worksheets)
Week 4:	Monday/Wednesday 10am, Tuesday/Thursday 10am (Worksheets)

Practicals

Week 4:	Friday 11am (not assessed)
Week 6:	Friday 11am (not assessed)
Trinity:	Week-Long Practical (assessed)

Reference Books (further references in the next slide)

HW	Wickham H (2014). Advanced R. Chapman & Hall.
CR	Crawley M (2012). The R Book. Wiley, 2nd edition.

Other Useful Books

General Programming

- Myers GJ, Sandler C and Badgett T (2012). The Art of Software Testing. Wiley.
- Hunt A and Thomas D (1999). The Pragmatic Programmer. Addison Wesley.
- McConnell S (2004). Code Complete. Microsoft Press.

Scientific Programming

- Venables WN and Ripley BD (2003). Modern Applied Statistics with S. Springer, 4th edition.
- Chambers J (2010). Software for Data Analysis: Programming with R. Springer.
- Spector P (2008). Data Manipulation with R. Springer.
- Maindonald J and Braun WJ (2003). Data Analysis and Graphics using R. Cambridge University Press, 3rd edition.

Scientific Data Visualisation

- Murrell P (2011). R Graphics. CRC, 2nd edition.
- Sarkar D (2008). **lattice**: Multivariate Data Visualization with R. Springer.
- Wickham H (2009). **ggplot2:** Elegant Graphics for Data Analysis.

Scientific Report Writing and Reproducible Research

- Gandrud C (2015). Reproducible Research with R and RStudio. CRC, 2nd edition.
- Xie Y (2015). Dynamic Documents with R and **knitr**. CRC, 2nd edition.

Overview

1. Statistical Computing: What Is It?
   Writing scientific software and data analysis pipelines, popular programming languages.
2. The R Programming Language
   Basic data types and structures. Importing and exporting data. Efficient vector computations. Graphics and figures. Notions of numeric optimisation and simulation.
3. Algorithms, Data Structures and Computational Complexity
   Pseudocode. Time and space complexity. Big-O notation.
4. Computational Architectures and Parallel Computing
   Distributing computations across computers. Parallel and sequential algorithms. Efficiency gains and overhead.
5. Testing, Debugging, Benchmarking, Profiling Code
   Good practices in software development. Function contracts and testing. Finding and fixing bugs. Performance assessment with benchmarking and profiling.
6. The Hadleyverse: **dplyr** & Co.
   Modern data manipulation tools: pipes and the **dplyr** package.
7. Literate Programming and Reproducible Research
   Documenting software and data analysis with literate programming.

Lecture Plan

L1: Statistical Computing: What is it?	P1: Worksheet 1
L2: Algorithms, Data Structures and Computational Complexity	P2: Worksheet 2
L3: Structuring Code: Files and Functions	P3: Worksheet 3
L4: Classes and Methods	P4: Worksheet 4
L5: Testing, Debugging, Benchmarking, Profiling Code	P5: Worksheet 5
L6: Computational Architectures and Parallel Computing	P6: Worksheet 6
L7: The Hadleyverse: **dplyr** & Co., Literate Programming and Reproducible Research	P7: Worksheet 7
L8: Distributing Code and Analyses: R Packages	P8: Worksheet 8

Statistical Computing: What is it?

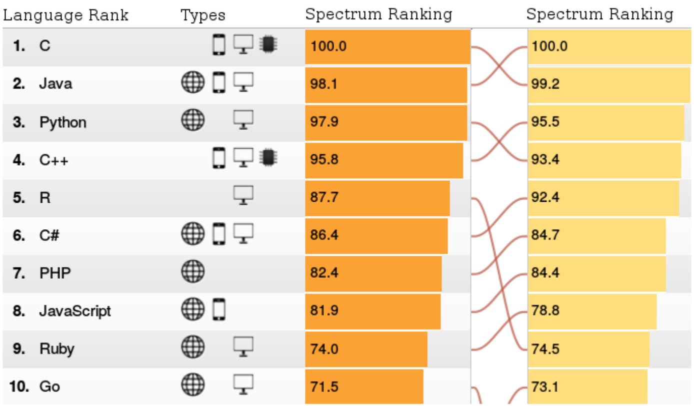
The Definition of Statistical Computing

Statistical computing deals with providing the computational tools for statistics (i.e. implementing software to do statistics on a computer) by using concepts and techniques from computer science.
The two main points we focus on are:

- the design and analysis of algorithms, to know what to expect from and how to implement an algorithm describing a statistical method;
- best practices in producing, documenting, assessing and distributing computer programs implementing statistical methods or the analysis of a specific data set.

The converse is computational statistics: the design of computationally intensive statistical methods that can only be used on computers regardless of the size of the problem. Some examples are the bootstrap, permutation tests, Markov chain Monte Carlo methods, etc.

And That is Why the Course Focuses on R



Language Rank	Types	Spectrum Ranking (2016)	Spectrum Ranking (2014)
1. C	📱 🖥 ▪	100.0	100.0
2. Java	🌐 📱 🖥	98.1	99.2
3. Python	🌐 🖥	97.9	95.5
4. C++	📱 🖥 ▪	95.8	93.4
5. R	🖥	87.7	92.4
6. C#	🌐 📱 🖥	86.4	84.7
7. PHP	🌐	82.4	84.4
8. JavaScript	🌐 📱	81.9	78.8
9. Ruby	🌐 🖥	74.0	74.5
10. Go	🌐 🖥	71.5	73.1

from IEEE Spectrum 2016 2014

(And that is why you should get familiar with Python as well...)

A Panoramic of Programming Languages

Choosing the right programming language for the job is crucial to solve it efficiently. The distinctions that matter the most in our case are:

- high-level languages (more human-readable, more difficult to optimise for performance) vs low-level languages (less human-readable, better control of implementation details);
- domain languages (geared towards a specific task, bad at anything else) vs general-purpose languages (which can do many things well, so you can build one whole software project without mixing languages);
- availability of libraries for statistics and machine learning (because we do not want to reimplement all kinds of models).

This leaves us with C/C++ for low-level high-performance code and R, Python for higher level code. R implements more models, but Python is more versatile (and catching up in the library department).

R: Pros & Cons

Pros:

- Thousands of packages, designed, maintained and widely used by statisticians, implementing most existing models and techniques.
- Very good at producing professional-quality plots and figures.
- Very flexible as a programming language, which is comparatively easy to code in; that in turn makes it possible to implement what is missing.

Cons:

- Thousands of packages designed and maintained by statisticians (who are not professional programmers, so the quality of those packages is often very bad). Also, no common naming convention.
- Very flexible as a programming language, which means often your code is not doing what you think it is doing.
- Numerical, analytical, optimisation methods are largely missing.

R: A Language and an Environment

R is both a language and an environment.

R as a language started as a clone of the S programming language, initially released in 1994 and again in stable form in 2000. R is high-level, interpreted language that is meant to act as the interface between the user and well tested C and Fortran libraries. It is still like that today: the only way to write efficient code is to leverage those libraries as much as possible, because R code can be very slow even on modern hardware.

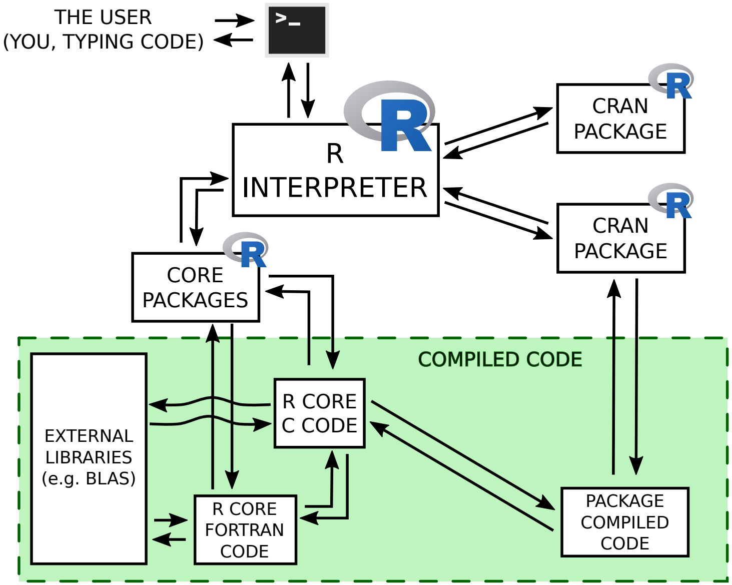
R as an environment provides a cross-platform terminal (available on all desktop operating systems) as well a distributed repository of packages, the Comprehensive R Archive Network (CRAN).

R is free software, as are almost all R packages.

Online Resources

Home Page:	`www.r-project.org`
	Home page of the R Project; this where you download R from.
CRAN:	`cran.r-project.org`
	The central repository of CRAN, where you download packages from.
GitHub:	`github.com`
	Many more packages here.
RSeek:	`www.rseek.org`
	A specialised search engine for content related to R, including packages, mailing list, blog posts, etc.
MetaCRAN:	`www.r-pkg.org`
	A parallel package repository that makes the source code of all packages accessible through a web interface.
Rstudio:	`www.rstudio.com`
	The only integrated development environment (IDE) explicitly designed for the R language.

The Architecture of the R Environment



Data Types and Vectors

All the interactions between different parts of R are built on the following atomic types (i.e. kinds of variables that are uniform and do not have internal structure)

- `integer`: `1L`, `2L`, `42L`, `...`
- `numeric`: `pi`, `0.25`, `+Inf`, `...`
- `logical`: `TRUE`, `FALSE`.
- `character`: `"lorem ipsum"`, `"a"`, `""`, `...`

NOTE:

- `integer` numbers are treated as `numeric` unless you postfix them with `L` (stands for "literal");
- all these types can also be `NA`, `NaN`, `NULL` (and you should take that in account when writing code);
- R does not really have scalar values, all variables are vectors; scalars are length-1 vectors and you can also have zero-length vectors, which tend to make your code crash.

Computers and Real Numbers: the Floating Point

A very, very common source of programming errors is misunderstanding how real numbers (`numeric`) are stored by computers: by using a floating point form like $\mathrm{tag} \times \mathrm{base}^{\mathrm{exponent}}$ with a decimal point positioned depending on the magnitude of the number.

Some fun facts about floating point arithmetic:

- computers are binary, so any number that cannot be expressed as a power(s) of 2 cannot be stored exactly and will be rounded;
- `numeric` values use 8 bytes of memory, which means numbers will be truncated to fit in $2^{64} - 1$ bits;
- the smallest representable number (which includes partial results like differences) is given by `sqrt(.Machine$double.eps)` $\approx 1.5 \times 10^{-8}$; smaller numbers underflow to zero ($\log$-scale is your friend);
- large numbers are not representable exactly, not even round numbers;
- look up "Why doesn't R think these numbers are equal?" on Google for more

Yes, It Happens in Real Life

```
sqrt(2) * sqrt(2) == 2
## [1] FALSE
(2^(0.5))^2 %% 1 == 0
## [1] FALSE
sqrt(2) * sqrt(2) - 2
## [1] 4.44e-16
```

```
0.1 + 0.1 + 0.1 == 0.3
## [1] FALSE
0.1 * 3 == 0.3
## [1] FALSE
```

```
1e99 == 1e99 + 1
## [1] TRUE
1 - 1e-20 == 1
## [1] TRUE
```

Operator Precedence

Together with floating point arithmetic, another very subtle source of confusion is operator precedence. R implements all the usual arithmetic (+, −, ∗, /, %%, ˆ), logical (|, ||, &, &&) and linear algebra operators (%∗%). In which order are they evaluated? Consider the following:

```
-2^{0.5}
## [1] -1.41
(-2)^{0.5}
## [1] NaN
-(2^{0.5})
## [1] -1.41
```

```
TRUE || FALSE == FALSE || FALSE
## [1] TRUE
(TRUE || FALSE) == (FALSE || FALSE)
## [1] FALSE
TRUE || (FALSE == FALSE) || FALSE
## [1] TRUE
```

Therefore:

- always add spaces around operators to make them stand out;
- do not be afraid to use parentheses to make statements unambiguous.

Data Structures: Vectors, Matrices and Arrays

As we noted above, both scalars and vectors are vectors in R.

Matrices (2 dimensions) and arrays ($3+$ dimensions) are also vectors; their columns are concatenated in a one-dimensional vector and the dimensions are saved as a separate attribute.

```
m = matrix(c(1.5, 2, 4.3, 0.9), ncol = 2, nrow = 2)
m
##      [,1] [,2]
## [1,]  1.5  4.3
## [2,]  2.0  0.9
dput(m)
## structure(c(1.5, 2, 4.3, 0.9), .Dim = c(2L, 2L))
```

As you can see the values of the cells are stored in column-major order. A side-effect of this representation is that all the value must be of the same atomic type (e.g. all `integer`, all `numeric`, etc.).

Data Structures: Factors

Another data structure that is fundamentally a vector with attributes are `factors`, which are used to store categorical and ordinal variables (finite set of values, represented as strings, possibly ordered).

```
rgb = factor(c("red", "blue", "green", "red", "green"))
levels(rgb)

## [1] "blue"  "green" "red"

dput(rgb)

## structure(c(3L, 1L, 2L, 3L, 2L), .Label = c("blue", "green",
## "red"), class = "factor")
```

R stores them as an integer vector (note the `L` notation) whose values are the indexes of the labels of the levels of the `factor`. The order of the levels is irrelevant for categorical variables, it matters only for ordinal variables.

Data Structures: Lists and Data Frames

We also data structures that can hold heterogeneous data: R provides the `list` and the `data.frame`. A list is a collection of arbitrary R objects (vectors, factors, other lists); it places no restriction on what it can hold.

```
ll = list(rgb, m, TRUE)
dput(ll)
```

```
## list(structure(c(3L, 1L, 2L, 3L, 2L), .Label = c("blue", "green",
## "red"), class = "factor"), structure(c(1.5, 2, 4.3, 0.9), .Dim = c(2L,
## 2L)), TRUE)
```

A data frame is simply a list of equal-length vectors, which can be of different types; it is the most common data structure to store real-world data for analysis.

```
df = data.frame(rgb, intensity = c(255L, 0L, 35L, 112L, 60L))
dput(df)
```

```
## structure(list(rgb = structure(c(3L, 1L, 2L, 3L, 2L), .Label = c("blue",
## "green", "red"), class = "factor"), intensity = c(255L, 0L, 35L,
## 112L, 60L)), .Names = c("rgb", "intensity"), row.names = c(NA,
## -5L), class = "data.frame")
```

Which One Should You Choose?

Sometimes the nature of the data dictates which data structure(s) you should use; sometimes you will have to choose yourself between several alternatives. That is an important choice: using the wrong data structure will make your code needlessly more complicated, use more memory, and make your code slower. Some considerations:

- If your code will include many linear algebra operations, use a matrix instead of a data frame; otherwise R will convert your data frame to a matrix and it use 100% more memory. (Interestingly, that is also true for a few other common methods such as computing covariance and correlation matrices).
- If you are not using much linear algebra, use data frames. Sometimes R decides it need to make a copy of an object when you modify it, but in the case of data frames it only copies the modified columns.
- Factors tend to make for faster code than character vectors.
- Integers make for faster code than numeric vectors, but the difference is tiny. (And R may convert them anyway and waste memory.)

Conclusions and Remarks

- R is one of the obvious choices for programming languages in machine learning and data science, and it is especially designed to be easy to use for that. The alternative is Python, which implements fewer models but is general-purpose.
- R is an interpreted programming language, which means that your code is parsed and interpreted each time you write it. It is built as a wrapper around time-tested C and Fortran libraries, and this is where its speed comes from.
- Beware of floating point arithmetic and operator precedence when writing code (not specific to R, it is the same in Python).
- R provides a number of simple (vectors, matrices, factors) and more complex (lists, data frames) data structures; using them appropriately is crucial to produce good code.

Algorithms, Data Structures and Computational Complexity

A Formal Description of an Algorithm: Pseudocode

An algorithm is a list of steps to be performed in order (thus defining a sequence of operations) to solve a problem. It is meant to be an unambiguous description of the steps you must go through to get to the solution (output) from the available information (input).

While algorithms are ultimately meant to be implemented in a programming language and to be executed by a computer, it is also useful to study them in abstract. In order to do that, we write them in pseudocode: an informal high-level description that uses the structural conventions of a normal programming language, but is intended for human reading rather than computer execution.

No standard for pseudocode syntax exists, and you will find several styles that vary from "almost identical to R code" to "almost identical to a free-form description".

An Example of Pseudocode

```
Harteminkʼs Information-Preserving Discretisation
Input: a data set $\mathbf{X} = X_i, i = 1, \ldots, p$ where all $X_i$ are continuous variables.
Output: a data set with $p$ discrete variables, each with $k_2$ levels.

1. Discretise each variable independently using quantile discretisation and a
   large number $k_1$ of intervals, e.g., $k_1 = 50$ or even $k_1 = 100$.
2. Repeat the following steps until each variable has $k_2 \ll k_1$ intervals,
   iterating over each variable $X_i$, $i = 1, \ldots, p$ in turn:
   2.1 compute
         $\mathrm{M}_{X_i} = \sum_{j \neq i} \mathrm{MI}(X_i, X_j);$
   2.2 for each pair $l$ of adjacent intervals of $X_i$, collapse them in a single
       interval, and with the resulting variable $X_i^*(l)$ compute
         $\mathrm{M}_{X_i^*(l)} = \sum_{j \neq i} \mathrm{MI}(X_i^*(l), X_j);$
   2.3 set $X_i = \operatorname{argmax}_{X_i(l)} \mathrm{M}_{X_i^*(l)}$.
```

Algorithm Analysis and Computational Complexity

Computational complexity is a branch of computer science that focuses on classifying computational problems according to their inherent difficulty:

- as a function of input size (sample size $n \to \infty$, number of variables $p \to \infty$);
- as a function of how many resources will be used, in particular time (e.g. CPU time spent) and space (e.g. RAM or hard disk use);
- on average (how long it will typically take) or in the worst case (how long it can possibly take).

This means that we can make educated guesses about how much computational resources an algorithm will take just from its specification; although the actual implementation details also play an important role. This is called algorithm analysis.

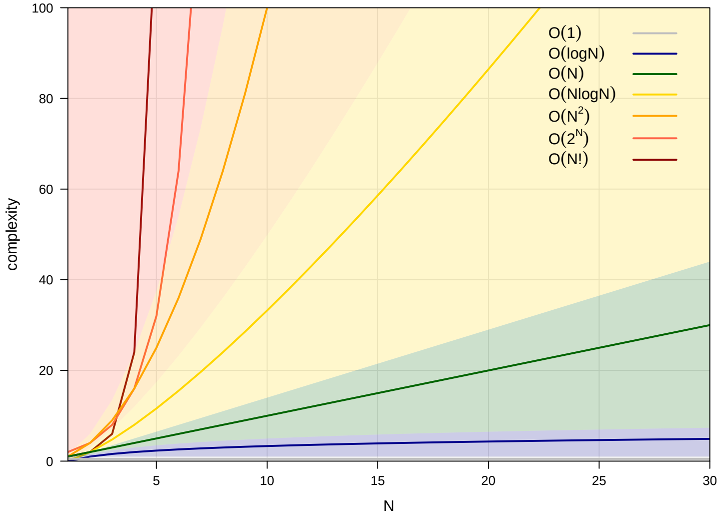
How Many Steps is it Going to Take?

The first and most important aspect in algorithm analysis is time complexity, that is, how many steps or how many operations the algorithm will take to complete. In order to do that we will simplify the analysis by assuming that:

- all operations take the same time, which we abstract with a theoretical complexity of 1;
- we only care about the class of computational complexity of algorithms, which we denote with the big-O notation, focusing on their behaviour for large inputs (all algorithms are fast with small input).

Big-O notation: an algorithm has a complexity $f(N) = O(g(N))$ if there exist positive constants n_0 and c such that $f(N) \leqslant c \cdot g(N)$, $\forall n > n_0$.

Classes of Computational Complexity



Up to $O(\mathrm{N})$ complexity is considered good, $O(\mathrm{N}\log\mathrm{N})$ is still OK, $O(\mathrm{N}^2)$ is feasible in practice depending on the problem, anything above $O(\mathrm{N}^2)$ is bad news.

Worst Case, Average Case, Best Case

More in detail, we may want to investigate more than just the average time complexity because the actual time complexity can vary by orders of magnitude depending on the nature of the input (for the same N). To do this we look at:

- The best case scenario, which we describe with the big-Omega notation: $f(N) = \Omega(g(N))$, $f(N) \geqslant c \cdot g(N)$. It represents a lower bound in time complexity.
- The average case, which we describe with the big-Theta notation: $f(N) = \Theta(g(N))$, $c_1 \cdot g(N) \leqslant f(N) \leqslant c_2 \cdot g(N)$. It represents the average time complexity.
- The worst case scenario, which we describe with the big-O notation.

In practice, this notation is usually completely disregarded and people just say things like "it is $O(g(N))$ on average and $O(h(N))$ in the worst case" using the big-O notation for all three cases.

Comparing Between Classes, Comparing Within Classes

How do we use the big-O notation to make comparisons?

- If we are comparing algorithms in different classes of complexity, can concentrate only on the leading term and ignore all other terms in $f(N)$ and even the leading coefficient:

$$O(3 \cdot 2^N + 3.42N^2) \gg O(2N^3 + 3N^2) \quad \text{becomes} \quad O(2^N) \gg O(N^3).$$

- If we are comparing algorithms in the same class of complexity, then we usually report the leading term and its coefficients, sometimes followed by the second largest term if it is relevant for the analysis:

$$O(1.2N^2 + 3N) \gg O(0.9N^2 + 2\log N) \quad \text{becomes } O(1.2N^2) \gg O(0.9N^2);$$
$$O(1.2N^2 + 3N) \gg O(0.9N^2 + 5N) \quad \text{stays the same.}$$

NOTE: all the examples here assume the size of the input can be summarised by a single parameter N, but often we need more (e.g. both sample size and number of variables). In that case, algorithms belong to different classes for different parameters (e.g. $O(NM^2)$).

Case Study: Fitting a Linear Regression Model

As you all know, the least squares/maximum likelihood estimate of the regression coefficients $\boldsymbol{\beta}$ in a linear regression model is

$$\underset{p\times 1}{\hat{\boldsymbol{\beta}}} = (\underset{p\times n}{X}^{T} \underset{n\times p}{X})^{-1} \underset{p\times n}{X}^{T} \underset{n\times 1}{\mathbf{y}}$$

and now we wonder, what is the time complexity of computing this estimate? Among the several ways of computing $\hat{\boldsymbol{\beta}}$, we will look into:

- using the closed-form formula above (the naive way);
- using the QR decomposition of X (how it is done in scientific software).

We characterise the size of the input using both n (i.e. diverging sample size) and p (i.e. high-dimensional samples).

Using the Closed-Form Formula

The steps we would perform if we were computing $\hat{\boldsymbol{\beta}}$ if we were doing it by hand are

1. compute $X^T X$;
2. take the result and compute $(X^T X)^{-1}$;
3. compute $X^T \mathbf{y}$;
4. multiply the results from steps 2 and 3.

From easily available sources (Wikipedia) we have that:

- multiplying an $r \times s$ matrix and an $s \times t$ matrix takes $O(rst)$ operations (multiplying and summing their cells);
- computing the inverse of an $r \times r$ matrix is $O(r^3)$ using a Cholesky decomposition or Gram-Schmidt.

Therefore, step 1 has time complexity $O(pnp) = O(np^2)$, step 2 is $O(p^3)$, step 3 is $O(np)$ and step 4 is $O(p^2)$. The overall time complexity is $O(np^2 + p^3 + np + p^2) = O(p^3 + (n+1)p^2 + np)$.

What Does That Mean? Can we do Better?

The interpretation of the overall time complexity is a follows:

- it is $O(p^3)$ in the number of parameters p
  or equivalently:
  if p doubles, it takes $8\times$ as long to compute;
- it is $O(n)$ in the sample size n
  or equivalently:
  if n doubles, it takes $2\times$ as long to compute.

You can match this theoretical complexity with the actual running time for on given n and p and you will know how long your computer will take to estimate $\hat{\boldsymbol{\beta}}$ for any value of n and p.

Can we do better? Only for extremely large matrices or by making more assumptions. For example, the Strassen algorithm performs matrix inversion in $O(r^{2.8})$ instead of $O(r^3)$ for $r > 1000$.

Using the QR Decomposition

Starting from the $X\boldsymbol{\beta} = \mathbf{y}$ formulation of the linear regression model, we can:

1. compute the QR decomposition of X (Q is $n \times p$, R is $p \times p$);
2. rewrite the problem as $R\boldsymbol{\beta} = Q^T\mathbf{y}$;
3. compute $Q^T\mathbf{y}$;
4. solve the resulting (triangular) linear system for $\boldsymbol{\beta}$.

For time complexity we have that:

- solving an $r \times s$ linear system with Gram-Schmidt is $O(rs^2)$;
- back-substitution to solve the system in step 4 is $O(s^2)$.

Therefore, step 1 is $O(np^2)$, step 3 is $O(np)$ and step 4 is $O(p^2)$; the overall time complexity is $O(np^2 + np + p^2) = O((n+1)p^2 + np)$. Again, we can do better (Gram-Schmidt + Householder transformations is $O(rs^2 - s^3/3)$ instead of $O(rs^2)$).

Comparing the Closed-Form Formula and QR

In conclusion:

- the closed-form formula for $\hat{\boldsymbol{\beta}}$ is $O(n)$ (linear in the sample size) and $O(p^3)$ (cubic in the number of regressors);
- The QR approach is again $O(n)$ but $O(p^2)$ (quadratic in the number of regressors).

Which algorithm is best depends on the data you expect to work on:

- if $n \to \infty$ but p is bounded, then it should not make much difference;
- if $p \to \infty$ then the QR approach will be faster (i.e. the time complexity is lower).

However, note that there are a number of additional considerations you should weight in choosing which algorithm to use: the matrix inverse in the closed-form formula is known to be numerically unstable, which is why the QR method is preferred in scientific software. And then there is space complexity...

A Second Kind of Complexity: Space Complexity

The second most important measure of complexity is space complexity, the number of memory cells which an algorithm needs. Unlike time complexity, it is well known how much memory each data type will use, so there is no need to define an abstract memory cell.

integers	real numbers	strings
4 or 8 bytes	4 to 10 bytes	1 to 4 bytes per character

Arguably, space complexity can actually be more important than time complexity: in principle, you can wait a bit longer to get the results, but if your program runs out of memory it will crash and you will get no results at all. Typically, there is a time-space tradeoff involved in a problem, that is, it cannot be solved with few computing time *and* low memory consumption. One then has to make a compromise and to exchange computing time for memory consumption or vice versa.

Case Study: Dense and Sparse Matrices

Matrices in which most cells are zero are called sparse matrices, as opposed to dense matrices in which most or all elements are non-zero. In the case of dense $(m \times n)$ matrices we need to store in memory the values of all the cells, which means the space complexity is $O(mn)$. However, in the case of sparse matrices we can just store the non-zero values and their coordinates, with the understanding that all other cells are equal to zero.

R provides several such representations in the **Matrix** package.

```
library(Matrix)
m = Matrix(c(0, 0, 2:0), 3, 5)
m
```

```
## 3 x 5 sparse Matrix of class "dgCMatrix"
##
## [1,] . 1 . . 2
## [2,] . . 2 . 1
## [3,] 2 . 1 . .
```

Comparing Space Complexity

How are the elements of m stored in memory? Here is how:

```
str(m)

## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots
##   ..@ i       : int [1:6] 2 0 1 2 0 1
##   ..@ p       : int [1:6] 0 1 2 4 4 6
##   ..@ Dim     : int [1:2] 3 5
##   ..@ Dimnames:List of 2
##   .. ..$ : NULL
##   .. ..$ : NULL
##   ..@ x       : num [1:6] 2 1 2 1 2 1
##   ..@ factors : list()
```

From the documentation, `i` contains the 0-based row numbers for each non-zero element in the matrix; `p` contains the index, for each column, of the initial (zero-based) index of elements in the column; and `x` contains the values of the non-zero cells.

Total space complexity: $O(3z)$ where z is the number of non-zero cells. In terms of real memory, $O(2z) * 4$ bytes $+O(z) * 8$ bytes $= O(16z)$ bytes.

What is the Trade-Off?

So, dense matrices have a space complexity of $O(8mn)$ bytes and sparse matrices have a space complexity of $O(16z)$ bytes; if $z \ll mn$ we can save most of the memory we would have used. What is the catch?

The catch is that operations may have a higher time complexity for sparse matrices than for dense matrices. Even the most simple: looking up the value of a cell (i, j) has time complexity $O(1)$ in a dense matrix, but for a sparse matrix we need to

1. look up what are the first and the last values in `x` for the column j by reading the jth element of `p`;
2. position ourselves on the row number for that first value in `i`, and read every successive number until we find the row number j or we reach the end of the column;
3. read the value of the cell from `x`, which has the same position in `x` as the row number in `i`; or return zero if we reached the end of the column.

Total time complexity: $O(1) + O(z/n) + O(1) = O(z/n)$ assuming z/n non-zero elements per column on average. Is $z/n \leqslant 1$?

Reading Can Take Time, Writing Can Take Time

Reading a cell from a sparse matrix is potentially more expensive than $O(1)$. Assigning a non-zero value to a cell, however, is always more expensive. For each of `i` and `x` we need to:

- allocate a new array of length $z + 1$;
- copy the z values from the old array;
- add the values corresponding to the new cell;
- replace the old array with the new one.

So time and space complexity both are $O(z)$. Handling `p` is also $O(z)$, and it is more complex since we will need to recompute half of the values on average.

This is troubling because we cannot predict the average time and space complexity just looking at the input size: they will change during the execution of our program as we assign new non-zero values to the sparse matrix.

Case Study: the Permutation Test for Zero Covariance

Consider the empirical covariance between two variables X and Y, with n observations each:

$$\mathrm{COV}(X,Y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

We can use a permutation test to compute the p-value for the hypothesis $H_0 : \mathrm{COV}(X,Y) = 0$ vs $H_1 : \mathrm{COV}(X,Y) \neq 0$ as follows:

1. compute $\gamma = \mathrm{COV}(X,Y)$;
2. for B times:
   - 2.1 create Y_b^* by permuting the elements of Y;
   - 2.2 compute $\gamma_b^* = \mathrm{COV}(X, Y_b^*)$;
3. compute the p-value as the proportion of $1/B \sum_{b=1}^{B} \mathbb{1}(|\gamma_b^*| > |\gamma|)$.

Do I need to compute this?

The time complexity of computing the covariance is $O(7n)$:

- $O(2n)$ to compute $\bar{x}$ and $\bar{y}$;
- $O(4n)$ to compute the numerator;
- $O(4n)$ to compute the denominator (some quantities are shared with the numerator).

Therefore, the overall time complexity of the algorithm is $O(10nB)$, and the space complexity is $O(B)$ to store the γ^*_b. We can get away with computing just

$$\mathrm{COV}(X, Y) \propto \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})$$

since the denominator is the same for (X, Y) and all (X^*, Y^*) and thus vanished in the comparison between γ and γ^*_b. Then time complexity is $O(2n)$ for computing the means $+O(3n)$ for the sum-product $= O(5n)$; space complexity is $O(1)$ to store the two means. To compute the p-value, we do this B times, so a semi-naive approach takes $O(5nB)$ time and $O(B)$ space.

Caching is Good!

If we privilege time complexity, we can precompute and cache

$$\tilde{x}_i = x_i - \bar{x} \qquad \text{and} \qquad \tilde{y}_i = y_i - \bar{y}$$

so that $\mathrm{COV}(X, Y) \propto \sum_{i=1}^{n} \tilde{x}_i \tilde{y}_i$ only takes $O(n)$ time $+O(2n)$ to compute the means $= O(3n)$ at the cost of having $O(2n)$ space complexity. ($\bar{y}$ is identical to $\bar{y}^*_b$, so we can permute the centred values directly.) In other words, we can trade one order of magnitude of space complexity for a $3.33\times$ speed up. Thus we have $O(3nB)$ and $O(2n + B)$ overall.

But we also care about space complexity; so we compute γ first and the we sum the $\mathbb{1}(|\gamma^*_b| > |\gamma|)$ as we compute them, instead of storing them; the variable we use for that is called an accumulator variable. This brings down space complexity from $O(2n + B)$ to $O(2n)$.

Summary and Remarks

- We can make a number of considerations on any algorithm before implementing it in R code, just looking at it description.
- First, we can formalise its steps, the operations it performs, the input data and the output using pseudocode.
- Then we can study it time and space complexity to see how it scales with the size of the input (and how much hardware we will need to run it).
- Finally, we can consider different forms and implementations of the same algorithm by trading time complexity for space complexity or vice versa. How the data are represented plays a key role in this.

Testing, Debugging, Benchmarking, Profiling Code

Software Quality Control

- Testing: verifying that code runs correctly by exercising the code under known conditions and checking that results are as expected. You do that during the process of writing the code, and when the code is complete. Definitely before using it!
- Debugging: discovering causes of incorrect behaviour and repairing them (repair is often easy, once you figure out the causes). This can happen either during testing or when actually using the code.
- Benchmarking: measuring performance for given tasks to check that it's within acceptable bounds. You should ideally do that on dummy data before using the code.
- Profiling: identifying performance bottlenecks and removing them to make overall program performance acceptable. You do that when you find out your program is too slow.

Function Contracts and Testing

Each function has an implied contract, which determines which arguments it accepts (and legal values for each) and what is its return value. A function fulfils its contracts if:

1. it returns the expected return value if all the arguments have legal values;
2. it returns an error if one or more arguments have illegal values;
3. (optional) it warns the user (with a self-explanatory message) if the values are legal but not necessarily meaningful, and tries to cope gracefully.

To ensure that is the case, we must set up a suite of tests in which each function in our program is called with different sets of legal and illegal arguments.

Example: the Truncated Normal

The density function of the truncated normal distribution is defined as

$$f(x; \mu, \sigma, l, u) = \frac{\frac{1}{\sigma}\phi(\frac{x-\mu}{\sigma})}{\Phi(\frac{u-\mu}{\sigma}) - \Phi(\frac{l-\mu}{\sigma})} \quad \text{in } [u, l] \text{ and zero elsewhere.}$$

It can be readily implemented in R using `dnorm()` and `pnorm()`.

```
truncated = function(x, mu, sigma, lower, upper) {

  dens = (dnorm(x, mu, sigma) / sigma) /
         (pnorm(upper, mu, sigma) - pnorm(lower, mu, sigma))

  dens[(x < lower) | (x > upper)] = 0

  return(dens)

}#TRUNCATED
```

What is the Contract of This Function?

From the mathematical definition, and taking into account the user's expectations, we have that:

- `x` should be a `numeric` vector, possibly of length zero.
- `mu` should be a `numeric` vector of length 1, which can assume any of the special values below.
- `sigma` should be a `numeric` vector of length 1 and `sigma >= 0`.
- both `lower` and `upper` should be `numeric` vectors of length 1 and `lower <= upper`.

All the arguments of `truncated()` should handle the special values `-Inf`, `+Inf`, `NA`, `NaN` and `NULL` appropriately.

We also require the return value to be a numeric vector of the same length as `x` containing only non-negative values (since it is a density function).

Different Types of Testing

Software testing falls into two categories:

- Unit testing means writing and running tests to exercise a well-defined portion (unit) of the code, including individual functions. You can and should test each unit as you're developing it.
- System testing (also known as functional or integration testing) involves running an entire program with known inputs. System testing comes afterwards, given that each module in the program works properly (passes unit tests), does the whole program work?

In both cases we want to take particular care and

- check outputs against known-good values from a different source;
- check boundary values values for each argument.

NOTE: here our whole program is a single function so the unit testing and system setting coincide, but that is generally not the case.

Implementing Tests in R

Tests are implemented in R with a combination of `try()` (to catch errors) and `stopifnot()` (to make the test fail if the desired condition is not met).

Example 1: `x` has an illegal (character) value. If we do

```
test = try({ truncated(x = "a", mu = 0, sigma = 1, lower = -2, upper = 2) })
```

then as expected `try()` says the result of the unit test is an error:

```
class(test)
## [1] "try-error"
```

That is exactly what should happen, so we use `stopifnot()` to make the unit test fail if `test` is not an error.

```
stopifnot(is(test, "try-error"))
```

Implementing Tests in R

Example 2: we check the boundary case `lower == upper`; we expect that the density should be a `+Inf` Dirac mass.

```
test = try({ truncated(x = 0, mu = 0, sigma = 1, lower = 0, upper = 0) })
test

## [1] Inf
```

`test` does not contain an error, but the expected return value from `truncated`. Hence we determine the outcome of the unit test with:

```
stopifnot(is.infinite(test) && (test > 0))
```

Example 3: we check the illegal case in which `lower > upper`.

```
test = try({ truncated(x = 0, mu = 0, sigma = 1, lower = 1, upper = 0) })
stopifnot(is(test, "try-error"))

## Error:  is(test, "try-error") is not TRUE
```

This is because we do not check for that (or, indeed, anything else) in the function. Now that the unit test has highlighted the problem, we know how to modify `truncated()` to match its contract.

Implementing Tests in R

Example 4: we know that if `mu = 1`, `sd = 1`, `lower = -2`, `upper = 2` then the density should assume the value 0.41913 at `x = 0.5`.

```
test = try({ truncated(x = 0.5, mu = 1, sigma = 1, lower = -2, upper = 2) })
stopifnot(test == 0.41913)

## Error:  test == 0.41913 is not TRUE
```

Is that a bug in `truncated()`? Actually not, the problem is in the comparison inside `stopifnot()`, which does not take into account floating point errors and the fact that our reference value is rounded at the 5th decimal. To check equality of two real numbers, always use `all.equal()` with an acceptable tolerance.

```
stopifnot(all.equal(test, 0.41913, tol = 10^-5))
```

This modified check correctly determines that the unit test succeeded.

How Many Tests, and How to Run Them?

The number of combinations of {valid, boundary, illegal} values for the arguments increases with combinatorial speed in the number of arguments; clearly we do not want to spend months writing unit and system tests! Fortunately, an extensive research on commercial software shows that testing 3-way interactions catches about 90% of defects, and 2-way interactions catch about 80%. Hence testing all {valid, boundary, illegal} for individual arguments (while all other arguments have legal values), and then testing pairs of {boundary, illegal} values (while all other arguments have again legal values) is a good compromise.

The correct way to run tests is isolate them from each other and run them in a clean R session. You can do that by

1. saving each unit test in a separate R file;
2. either `source()`ing each file in an empty, throw-away environment with:

   ```
   source("test0001.R", local = new.env())
   ```

   or directly from a command line with Rscript.

Debugging

Incorrect behaviour can manifest itself in three ways:

- your program exits with an error message;
- your program prints a warning, and you later find the results it generated make no sense;
- your program seems to work correctly but the results make no sense.

In the first two cases we have some clue as to where the problem may be, in the last case you have to find out from scratch.

1. Realise you have a bug: that is one of the reasons for having an extensive unit and system test suite.
2. Make it repeatable: because you will need to reproduce it repeatedly to isolate it (e.g. save the random seed).
3. Figure out where it is: that is what we are covering next.
4. Fix it and test it: add a unit/system test to your suite that triggers the bug with the original code and does not with the fixed code.

Truncated Normal: Maximum Likelihood Estimates

```
truncated = function(x, mu, sigma, lower, upper) {

  dens = (dnorm(x, mu, sigma) / sigma) /
         (pnorm(upper, mu, sigma) - pnorm(lower, mu, sigma))

  dens[(x < lower) | (x > upper)] = 0

  return(dens)

}#TRUNCATED

truncated.loglikelihood = function(x, par)
  sum(log(truncated(x, mu = par["mu"], sigma = par["sigma"],
            lower = par["lower"], upper = par["upper"])))

maxlik = function(x, mu, sigma, lower, upper) {

  optim(par = c(mu = mu, sigma = sigma, lower = lower, upper = upper),
    fn = truncated.loglikelihood, x = x, method = "BFGS")

}#MAXLIK
```

Debugging from an Error Message

So, let's generate some random observations from a truncated normal and compute the corresponding maximum likelihood estimates of the parameters.

```
x = rnorm(1000, mu = 0, sigma = 1)
x = x[x > -4 & x < 4]

maxlik(x, mu = 0, sigma = 1, lower = -2, upper = 2)
## Error in optim(par = c(mu = 0, sigma = 1, lower = -2, upper = 2), fn =
truncated.loglikelihood, :  initial value in 'vmmin' is not finite
```

The maximisation fails, with an error message from `optim()`. `traceback()` prints the backtrace (!) of function calls leading to the (last encountered) error.

```
traceback()
## 2: maxlik(x, mu = 0, sigma = 1, lower = -2, upper = 2)
## 1: optim(par = c(mu = 0, sigma = 1, lower = -2, upper = 2),
##        fn = truncated.loglikelihood, x = x, method = "BFGS")
```

Interactive Debugging

Since the error message is not clear on the exact nature of the problem, we need to investigate what is going on with the `optim()` call inside `maxlik()`. To do that, we set

```
options(error = recover)
```

which tells R to open an interactive debug prompt when an error occurs. (The default action is to stop and print the error message.)

```
maxlik(x, mu = 0, sigma = 1, lower = -2, upper = 2)
## Error in optim(par = c(mu = 0, sigma = 1, lower = -2, upper = 2), fn =
truncated.loglikelihood, :  initial value in 'vmmin' is not finite
Enter a frame number, or 0 to exit
1: maxlik(x, mu = 0, sigma = 1, lower = -2, upper = 2)
2: #3: optim(par = c(mu = 0, sigma = 1, lower = -2, upper = 2), fn = ...

Selection: 1
Called from: top level
Browse[1]> ls()
## [1] "lower" "mu"    "sigma" "upper" "x"
```

Peeking Inside Functions

Now that we have an interactive debug prompt open inside `maxlik()`, we can check a number of things. First, that the arguments that hold the initial values have the values we specified. (That may not obviously be the case when we have many nested function calls, and each sanitised its arguments.)

```
Browse[1]> c(mu, sigma, lower, upper)
## [1]  0  1 -2  2
```

Then, since the error message mentions `initial value` and `not finite`, we check the return value `truncated.wrapper()` and we find out that it is `-Inf`.

```
Browse[1]> truncated.loglikelihood(x, par = c(mu = mu, sigma = sigma,
             lower = lower, upper = upper))
## [1] -Inf
```

Fixing the (First) Bug

A log-likelihood of `-Inf` means a likelihood of zero for at least one `x`; and that can happen only if `x` is above `upper` or below `lower`.

```
range(x)
## [1] -3.49  3.20
```

Now that we have identified (a first) problem, we can fix it; one option is to set the likelihood to have a small positive value outside `lower` and `upper`. The value should be so small that `optim()` cannot possibly choose it as the maximum but large enough that it is not rounded to zero: the usual choice is `sqrt(.Machine$double.eps)` $\approx 1.5 \times 10^{-8}$. Therefore:

```
truncated = function(x, mu, sigma, lower, upper) {

  dens = (dnorm(x, mu, sigma) / sigma) /
         (pnorm(upper, mu, sigma) - pnorm(lower, mu, sigma))

  dens[(x < lower) | (x > upper)] = sqrt(.Machine$double.eps)

  return(dens)

}#TRUNCATED
```

Is That All? More Bugs to Come

Now `maxlik()` completes without any error, but the parameter estimates it returns are nonsensical; there are still bugs for us to fix.

```
maxlik(x, mu = 0, sigma = 1, lower = -2, upper = 2)$par

##       mu    sigma    lower    upper
##      513   171393 -454908   455513
```

Going by elimination, `truncated()` should now be correct so the problem should be either in `truncated.loglikelihood()`, `maxlik()` or both. We now need to modify our code to print enough diagnostic messages (often called tracing statements) to establish how the optimization is carried out.

```
maxlik = function(x, mu, sigma, lower, upper) {

  optim(par = c(mu = mu, sigma = sigma, lower = lower, upper = upper),
    fn = truncated.loglikelihood, x = x, method = "BFGS",
    control = list(trace = 2))

}#MAXLIK
```

Drilling Down into the Code

Thanks to the `trace` argument, we find out a surprising fact: the (log)likelihood decreases at every iteration! `optim()` returns the minimum, not the maximum (as noted in the documentation).

```
maxlik(x, mu = 0, sigma = 1, lower = -2, upper = 2)$par
## initial  value -2160.305907
## iter  10 value -15087.258331
## iter  20 value -20754.663359
## final  value -25014.436139
## converged
##       mu   sigma   lower   upper
##      513  171393 -454908  455513
```

To find the maximum then we need to change the sign of the loglikelihood in `truncated.loglikelihood()`.

```
truncated.loglikelihood = function(x, par)
  -sum(log(truncated(x, mu = par["mu"], sigma = par["sigma"],
            lower = par["lower"], upper = par["upper"])))
```

One More Time!

```
maxlik(x, mu = 0, sigma = 1, lower = -2, upper = 2)$par
## initial  value 2160.305907
## iter  10 value 1443.851256
## iter  20 value 1323.314194
## final  value 1323.166567
## converged
##       mu    sigma    lower    upper
##  -0.0326   0.7424 -50.7471   3.9833
```

Now the estimates of `mu` and `sigma` are more or less on target, as is `upper`; but `lower` is still completely wrong. Again there are no error messages, and the tracing messages look fine. Going by exclusion again, the only argument of `optim()` we have not touched yet is `method`, which determines which numeric optimization algorithm will be used. Maybe we should try another one.

```
maxlik = function(x, mu, sigma, lower, upper) {

  optim(par = c(mu = mu, sigma = sigma, lower = lower, upper = upper),
    fn = truncated.loglikelihood, x = x, method = "Nelder-Mead")

}#MAXLIK
```

Finally!

At long last, we now have maximum likelihood estimates that are close to the parameter values we used to generate the data.

```
maxlik(x, mu = 0, sigma = 1, lower = -2, upper = 2)$par

##      mu   sigma   lower   upper
## -0.0327  0.7424 -3.6266  3.5094
```

To make debugging easier and faster, in your code you should:

- use clear error and warning messages;
- in the case of complex functions, add a `trace` or `debug` argument that activates tracing statements;
- sanitise all arguments and not rely on the functions you call to do so, so that the location of the errors is indicative;
- possibly check that the return value is legal before returning it.

How Can You Debug Warning Messages?

You can turn all warnings into errors by setting the appropriate global option:

```r
options(warn = 2)
```

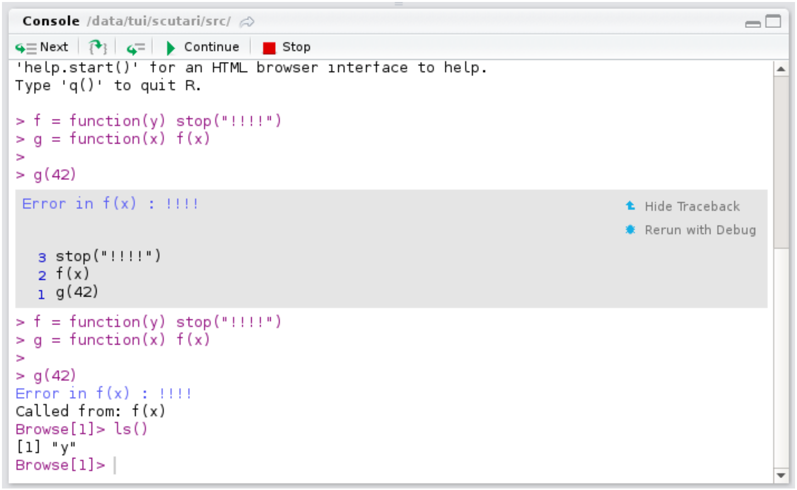
Then you can proceed to debug using the same techniques we have just seen.

Possible values for this option are:

- less than zero: all warnings are ignored.
- zero (the default): warnings are stored until the top-level function returns.
- one: warnings are printed as they occur.
- two or larger: all warnings are turned into errors.

See more in `?options`.

Debugging in RStudio



```
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> f = function(y) stop("!!!!")
> g = function(x) f(x)
>
> g(42)
Error in f(x) : !!!!

  3 stop("!!!!")
  2 f(x)
  1 g(42)
> f = function(y) stop("!!!!")
> g = function(x) f(x)
>
> g(42)
Error in f(x) : !!!!
Called from: f(x)
Browse[1]> ls()
[1] "y"
Browse[1]>
```

(It works in the same way, but you click instead of calling R commands.)

Benchmarking Running Times: `system.time()`

The most appropriate way of measuring the performance of a piece of R code depends on how long it takes to run; but in most real-world cases (running times $\geqslant 60$s) you can use `system.time()` to benchmark it.

```
x = rnorm(10^5, 0, 1)
x = x[x > -4 & x < 4]
system.time(replicate(100,
  maxlik(x, mu = 0, sigma = 1, lower = -2, upper = 2) )) / 100

##    user  system elapsed
## 2.06584 0.00163 2.06788
```

The return value contains three values:

- user: time used by R;
- system: time used by the operating system (reading and writing files to disc, allocating memory, sending data over the network, etc.);
- elapsed: the total running time (often called wall clock time).

NOTE: compute the time of several runs and then divide it by the number of runs, for better precision.

UNIVERSITY OF OXFORD

Benchmarking Short Running Times: **microbenchmark**

`system.time()` is too inaccurate to benchmark small bits of code that take only a few seconds to run. The **microbenchmark** package uses experimental design techniques to give timings with a precision of 1 millisecond (10^{-3}s).

```
library(microbenchmark)

df = as.data.frame(matrix(0, nrow = 1000, ncol = 1000))
m = numeric(ncol(df))
microbenchmark(
  COLMEANS = m <- colMeans(df),
  APPLY = m <- apply(df, 1, mean),
  SAPPLY = m <- sapply(df, mean),
  LOOP = for (i in seq(ncol(df))) m[i] <- mean(df[, i])
)

## Unit: milliseconds
##      expr   min    lq  mean median    uq   max neval  cld
##  COLMEANS  4.95  5.16  7.03   5.70  6.65 44.31   100  b
##     APPLY 28.00 61.75 60.89  63.42 66.11 70.01   100    d
##    SAPPLY  3.90  4.12  4.32   4.27  4.48  5.44   100 a
##      LOOP 10.99 11.38 11.85  11.59 12.19 14.88   100   c
```

(Because loops can be faster than `apply()` if you misuse it.)

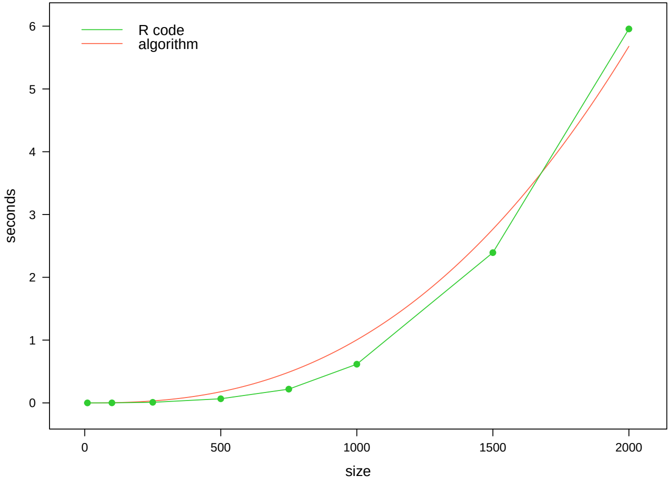
Benchmarking to Check Computational Complexity

Benchmarking is also useful to check the computational complexity of the R code implementing an algorithm; ideally, it should be the same as the computational complexity of the theoretical algorithm. If the two differ, our code is inefficient and we should profile it to find the reason.

For instance, matrix multiplication has a computational complexity $O(N^{2.5})$ for $N \times N$ matrices.

```
size = c(10, 100, 250, 500, 750, 1000, 1500, 2000)
timing = numeric(length(size))

for (s in seq_along(size)) {

  X = matrix(0, nrow = size[s], ncol = size[s])
  timing[s] = system.time(replicate(100, t(X) %*% X))["elapsed"]/100

}#FOR
```

Algorithm Complexity, R Code Running Time



The computational complexity of the R code is indeed $O(N^{2.5})$, or more precisely `seconds = 3.174e-08 * size^2.5`.

Profiling

Consider the `microbenchmark()` example once more. Why is `apply()` is slow? We can find out by looking for parts of the code that take time to run. Keep in mind that:

- producing a correct program is more important than producing a fast program;
- 20% of the code will consume 80% of the overall running time;
- the number of lines of code has no relationship with the overall running time;
- expectations about performance are often wrong - do not assume, measure!
- it is almost impossible to identify performance bottlenecks before a program is complete - write for clarity and profile at the end!

NOTE: we concentrate on execution speed, but it is important to profile for memory use as well.

The Old-Fashioned Way: Profiling with Rprof()

```
df = as.data.frame(matrix(0, nrow = 1000, ncol = 1000))
profile = tempfile()
Rprof(file = profile)
for (i in 1:1000) apply(df, 1, mean)
Rprof(NULL)
lapply(summaryRprof(profile)[c("by.self", "by.total")], head, n = 4)
unlink(profile)
```

```
## $by.self
##                   self.time self.pct total.time total.pct
## "apply"               11.64    38.72      30.06    100.00
## "unlist"               7.58    25.22       7.58     25.22
## "aperm.default"        4.88    16.23       4.88     16.23
## "mean.default"         1.90     6.32       2.06      6.85
##
## $by.total
##                          total.time total.pct self.time self.pct
## "apply"                       30.06    100.00     11.64    38.72
## "as.matrix.data.frame"        10.02     33.33      0.60     2.00
## "as.matrix"                   10.02     33.33      0.00     0.00
## "unlist"                       7.58     25.22      7.58    25.22
```

The New Fashionable Way: Profiling with **profvis**

A more modern approach to profiling is implemented in the **profvis** package, which is also well integrated in RStudio. Its provides a `profvis()` function which works like `Rprof()`.

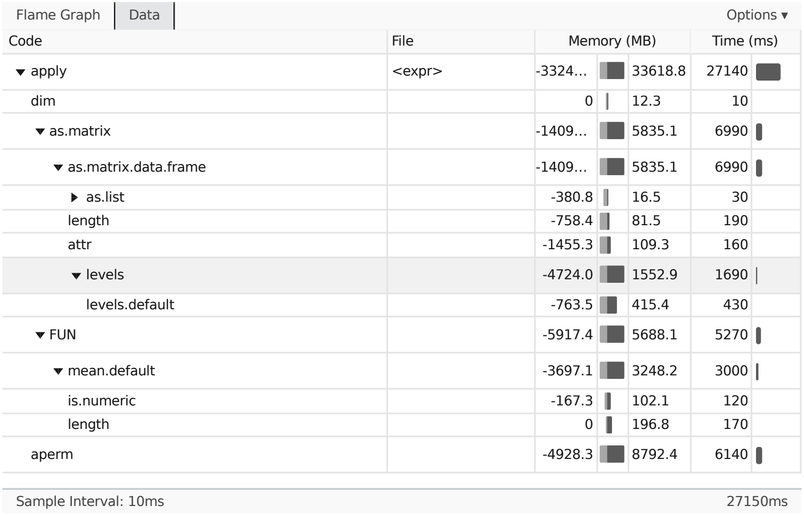
```r
library(profvis)

profvis({ for (i in 1:1000) apply(df, 1, mean) })
```

(It is preferable to use a `for` loop instead of `replicate()` because that would show up in the profiling.)

The main advantages of `profvis()` are that:

- provides a graphical display of the profiling, either embedded in RStudio or in a separate browser window (much like manual pages);
- profiles memory usage as well as running times.

Profiling with **profvis**



Code	File	Memory (MB)		Time (ms)
▾ apply	<expr>	-3324…	33618.8	27140
dim		0	12.3	10
▾ as.matrix		-1409…	5835.1	6990
▾ as.matrix.data.frame		-1409…	5835.1	6990
▸ as.list		-380.8	16.5	30
length		-758.4	81.5	190
attr		-1455.3	109.3	160
▾ levels		-4724.0	1552.9	1690
levels.default		-763.5	415.4	430
▾ FUN		-5917.4	5688.1	5270
▾ mean.default		-3697.1	3248.2	3000
is.numeric		-167.3	102.1	120
length		0	196.8	170
aperm		-4928.3	8792.4	6140

Sample Interval: 10ms 27150ms

Summary and Remarks

- Writing code is only part of statistical programming.
- You should check that your code actually does what it is meant to do (testing).
- You should investigate bugs and fix them - and then add more tests to make sure they stay fixed (debugging).
- You should make sure your programs run fast enough to meet deadlines and to handle large data sets (benchmarking & profiling).
- Last but not least, you should consider developing any kind of scientific software as cycle: planning $\rightarrow$ analysis $\rightarrow$ design $\rightarrow$ implementation $\rightarrow$ testing $\rightarrow$ and back again for changes in scope and improvements.

Computational Architectures and Parallel Computing

Parallel Computing: Hardware

Parallel computing is defined as the execution of several calculations simultaneously; this is one of the most common ways to speed up computations, especially when we cannot define new algorithms with a better computational complexity. It can be implemented as a combination of hardware, software and algorithms. Hardware architectures are classified as:

- Single-Instruction, Single-Data (SISD): a single processing unit performing a single operation on the same data (e.g. old single-processor single-core computers).
- Multiple-Instruction, Single-Data (MISD): multiple processing units performing different operations (independently and asynchronously) on the same data.
- Single-Instruction, Multiple-Data (SIMD): multiple processing units performing the same operation on multiple data (e.g. GPUs).
- Multiple-Instruction, Multiple-Data (MIMD): multiple processing units performing different operations on multiple data (e.g. multi-core CPUs, distributed computing).

Parallel Computing: Software

As for the software, the most important characteristics are:

- who is performing the computations:
    - the R process you are using;
    - completely independent R processes (possibly running on different computers);
    - processes accessing the same session;
    - threads within the same process (not really an option with R);
- how data is handled:
    - only one copy (shared memory, or remote database);
    - different copies for different processes/threads;
- how information is passed around:
    - over a network connection;
    - local inter-process communication;
    - shared memory.

Common Scenarios for Hardware and Software

- Your Laptop
  - Your CPU has a few cores (say 2, 4); you can use all but one, which should be kept free for your operating system to use.
  - You have limited memory (say 4GB), so having multiple copies of large data sets around is not an option.
- A Compute Server
  - More CPU power (say 32-64 cores) and more memory (128GB to 1TB).
  - Typically data is read from and written to a separate server (with lots of disk space) over the network, so importing and exporting data is slow but moving it between processes is fast.
  - All copies of the data are stored in the same pool of memory.
- Distributed Computing
  - A number of servers (say, 20) each with a number of cores (8-16) and decent memory (64GB to 128GB).
  - Moving data to and from the servers (over the network) is slow, but memory usage, disk usage and CPU load can be spread better.

Parallel Computing: Algorithms

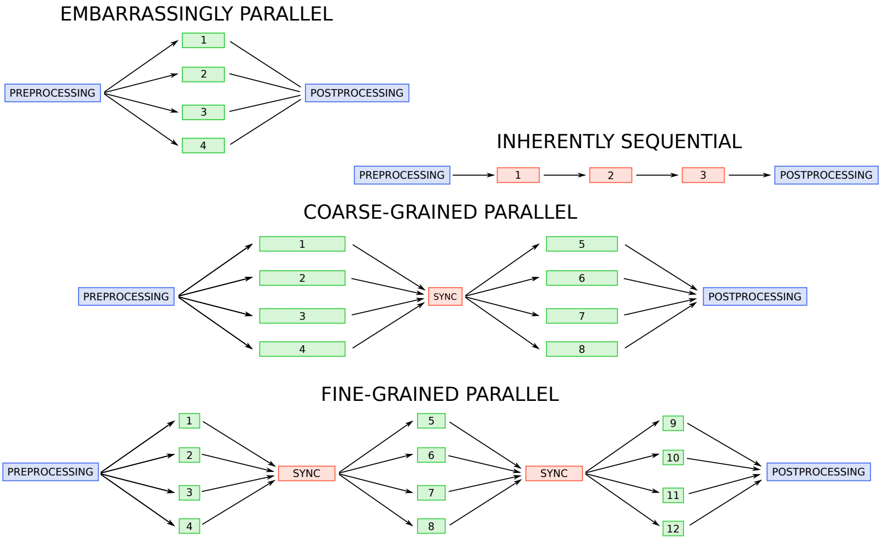
It is important to note that the degree to which an algorithm can leverage parallel processing depends on the nature of the problem it is trying to address.

Some problems are embarrassingly parallel, that is, they can be split in such a way that each part never needs to exhange information with other parts.

Other problems cannot be fully parallelised, because their parts have to communicate periodically with each other to synchronise their state. If frequent synchronisations are required we speak of fine-grained parallelism, and of coarse-grained parallelism if synchronisations are only needed a few times over a long period of time.

Finally, some problems are inherently sequential, and cannot be parallelised at all.

Parallel and Sequential Algorithms



A Reality Check: Overhead

Ideally, we would expect that if

- an algorithm is embarrassingly parallel; and
- it takes $T_{\text{SEQUENTIAL}}$ time to complete when performing all steps sequentially;

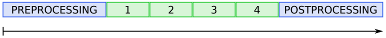
with a parallel implementation that uses M processes it would take

$$T_{\text{IDEAL}}(M) = \frac{T_{\text{SEQUENTIAL}}}{M}$$

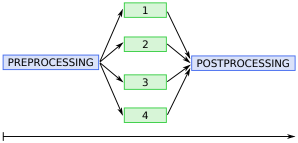
whereas in practice it will take $T_{\text{REAL}}(M) = T_{\text{IDEAL}}(M) + T_{\text{OVERHEAD}}(M)$ with $T_{\text{OVERHEAD}}(M)$ increasing in M. There are several reasons for that:

- the steps of the algorithms that are executed in parallel take different times to complete, so we must wait for the one that takes the longest;
- communication overhead (it takes time to send the data and collect the results);
- bottlenecks (hard disks, network connections are shared between multiple processes and can be saturated);
- if $M \gg$ number of cores, not all processes can run at the same time.

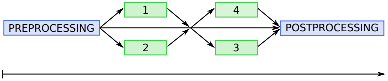
Two Examples of Overhead



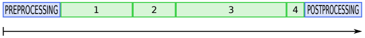
$T_{\text{SEQUENTIAL}}$



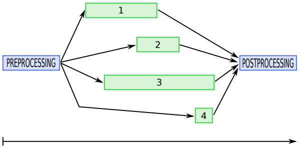
$T_{\text{REAL}}(4)$



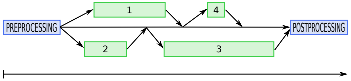
$T_{\text{REAL}}(2)$



$T_{\text{SEQUENTIAL}}$



$T_{\text{REAL}}(4)$



$T_{\text{REAL}}(2)$

The Law of Diminishing Returns

Eventually, as we add more processes, $T_{\text{REAL}}(M+1) \geqslant T_{\text{REAL}}(M)$ since

$$\frac{T_{\text{SEQUENTIAL}}}{M+1} + T_{\text{OVERHEAD}}(M+1) \geqslant \frac{T_{\text{SEQUENTIAL}}}{M} + T_{\text{OVERHEAD}}(M)$$

simplifies to

$$\frac{T_{\text{SEQUENTIAL}}}{M(M+1)} \leqslant T_{\text{OVERHEAD}}(M+1) - T_{\text{OVERHEAD}}(M)$$

which holds for large enough M as the left-hand converges to zero, but the right-hand side generally does not.

This is called the law of diminishing returns: adding more and more processes results in smaller and smaller gains and it eventually makes things slower.

A Remedy for Computational Complexity?

Another point that is worth noting is that parallel computing is not a remedy for high computational complexity because:

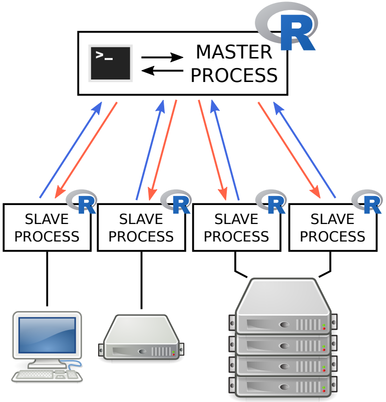
- the size N of the input can grow and grow, but the number of available processor cores is fixed (and that in turn limits M as well);
- if the computational complexity of the algorithm is larger than $O(N)$, even adding 1 new process for each new input would not prevent longer running times;
- the law of diminishing returns limits the number of processes which we can use effectively.

Parallel Computing in R: the **parallel** Package

The standard solution to perform parallel computing in R is the **parallel** package (which incorporated the old **snow** and **multicore** packages).

parallel implements a master-slave setup in which the user works on the (interactive) master process; and the master process in turn distributes the data and the R commands to a cluster of (non-interactive) slave processes (blue arrows), to collect the results when the slaves are done (red arrows).

The master process should just be used to coordinate the slaves, without performing much in the way of computations.



Basic Usage and Functions: Independent Slave Processes

Basic Workflow:

1. load the **parallel** package.

```
library(parallel)
```

2. create a cluster of 4 slaves.

```
cl = makeCluster(4)
```

3. load packages in the slaves.

```
ignore = clusterEvalQ(cl, library(MASS))
```

4. work in parallel on the slaves.

```
rand = clusterCall(cl, runif, n = 10^4)
```

5. work on the results.

```
sapply(rand, mean)
## [1] 0.499 0.499 0.499 0.506
```

6. stop the cluster to kill the slaves.

```
stopCluster(cl)
```

Relevant Functions:

`parApply()`, `parSapply()`, `parCapply()`, `parRapply()`, `parLapply()`

Parallel functions that work like the applied functions in base R, but distribute the computation to the slaves.

`clusterCall()`, `clusterEvalQ()`

Functions that evaluate an arbitrary expression or call an arbitrary function on each slave.

`clusterExport()`

Copy R objects from the master to the slaves.

`parLapplyLB()`, `parSapplyLB()`, `clusterCallB()`

Same as the equivalent functions above, but with better load balancing between the slaves.

Different Types of Clusters

The **parallel** package gives you several options for creating clusters, each with different pros and cons.

- `SOCK` (**snow**, **parallel**) and `PSOCK` (**parallel**, the default)

  Independent slave processes that communicate with system (pipe) sockets (`PSOCK`) or network sockets (`SOCK`). All R objects must be copied to the slaves, but the slaves can run on different computers.

- `FORK` (**parallel**, not available on Windows)

  Slave processes are initialised with shared memory, and R objects are only copied if the slaves modify them. All slaves must run on the same computer.

- `MPI` (**parallel**)

  Slave processes are started by the MPI library, and behave in the same way as `SOCK` ones. However, the MPI library provides control over slaves at a much lower level for both execution and communicating with the master process.

NOTE: we do not cover threads, they are supported by **parallel** and **multicore**.

K-Means with Random Starts (Embarrassingly Parallel)

A classic example of parallel computing is k-means with random starts; each initial cluster label allocation is chosen randomly and independently.

```r
library(parallel)

cl = makeCluster(3, type = "PSOCK")
invisible(clusterEvalQ(cl, library(MASS)))
clustering =
  clusterEvalQ(cl, kmeans(Boston, centers = 4, nstart = 200))
total.ss = sapply(clustering, `[[`, "tot.withinss")
best = clustering[[which.min(total.ss)]]
stopCluster(cl)
```

In this case no data or functions are passed between the master and the slaves; both the **MASS** package and the `Boston` data set it contains are loaded directly in the slaves. Each slave sends back the return value of `kmeans()` to `clusterEvalQ()` in the master, which collects them and returns them in a list. Finally, we pick the set of clusters with the minimum "total within-cluster sum of squares" among them.

Nonparametric Bootstrap (Embarrassingly Parallel)

Another embarrassingly parallel method is nonparametric bootstrap:

1. we independently sample with replacement `R` bootstrap samples of the size as the data;
2. we compute a statistic of interest on each bootstrap sample;
3. we compute the mean (and often the standard deviation) of the resulting statistics.

The R code to do that (sequentially):

```
library(boot)

x = rexp(2, n = 10^6)
med = function(data, indices) median(data[indices])
system.time({ boot(x, med, R = 200) })
```

```
##    user  system elapsed
##  15.637   0.331  15.969
```

Nonparametric Bootstrap (Embarrassingly Parallel)

First, we can rework the same code to use `sapply()` (still sequential).

```
indices = replicate(200, sample(10^6, 10^6, replace = TRUE), simplify = FALSE)
system.time({ mean(sapply(indices, med, data = x)) })
```

```
##    user  system elapsed
##  10.533   0.358  10.891
```

Then, calling `parSapply()` instead of `sapply()` gives us the corresponding parallel implementation.

```
cl = makeCluster(2, type = "PSOCK")
system.time({ mean(parSapply(cl, indices, med, data = x)) })
```

```
##    user  system elapsed
##   0.897   0.082   7.537
```

```
stopCluster(cl)
```

Note that if $T_{\text{SEQUENTIAL}} \approx 11$ then $T_{\text{IDEAL}} \approx 11/2 = 5.5$ but $T_{\text{REAL}}(2) \approx 7.5$. So $T_{\text{OVERHEAD}}(2) \approx 7.5 - 5.5 \approx 2$. Often they are reported as

$$\frac{T_{\text{REAL}}(2)}{T_{\text{SEQUENTIAL}}} \approx 0.68 \qquad \text{in which} \qquad \frac{T_{\text{OVERHEAD}}(2)}{T_{\text{SEQUENTIAL}}} \approx 0.18.$$

When Overhead Becomes Too Much

Increasing the number of slaves from 2 to 4 makes performance better even though overhead increases as well; there is no benefit in using 8.

```
cl = makeCluster(4, type = "PSOCK")
system.time({ mean(parSapply(cl, indices, med, data = x)) })
##    user  system elapsed
##   0.954   0.105   6.342
stopCluster(cl)
```

$$\frac{T_{\text{REAL}}(4)}{T_{\text{SEQUENTIAL}}} \approx 0.5 \qquad \text{in which} \qquad \frac{T_{\text{OVERHEAD}}(4)}{T_{\text{SEQUENTIAL}}} \approx 0.25.$$

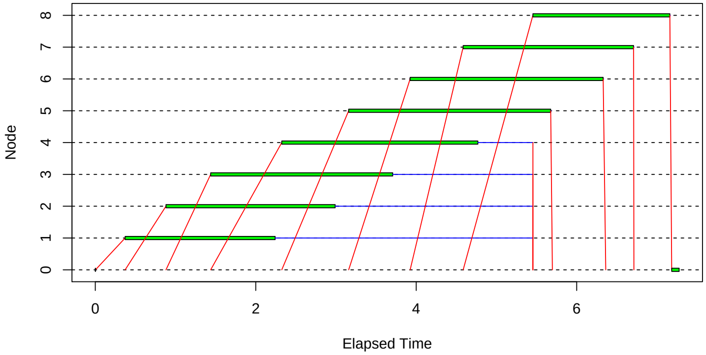
```
cl = makeCluster(8, type = "PSOCK")
system.time({ mean(parSapply(cl, indices, med, data = x)) })
##    user  system elapsed
##   1.172   0.157   4.908
stopCluster(cl)
```

$$\frac{T_{\text{REAL}}(8)}{T_{\text{SEQUENTIAL}}} \approx 0.5 \qquad \text{in which} \qquad \frac{T_{\text{OVERHEAD}}(8)}{T_{\text{SEQUENTIAL}}} \approx 0.375.$$

Investigating Overhead with the **snow** Package

If we re-run the same commands on a cluster created with **snow**, we can get a nice plot from `snow.time()` and look into our parallel performance.

```r
library(snow)
cl = snow::makeCluster(8)
timings = snow.time({ mean(snow::parSapply(cl, indices, med, data = x)) })
snow::stopCluster(cl)
```



What Went Wrong? And How to Fix It?

There seems to be two problems with our code:

- the master can copy the data to a single slave at a time, so some slaves spend too much time waiting;
- we are copying too much data, it appears it takes about 0.5 seconds each time which is about 8% of the total running time! The last slave starts working after $0.5 \times 8 = 4$ seconds out of 6.

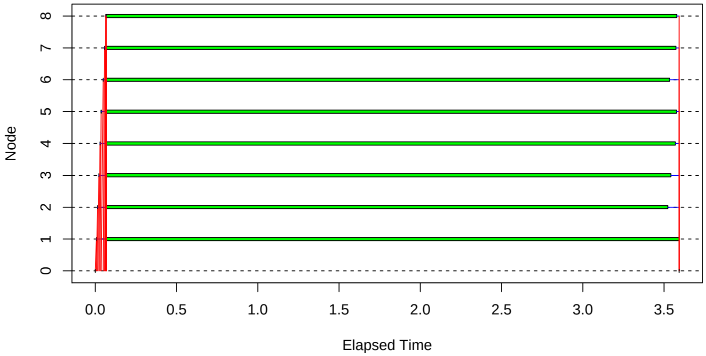
We can address both problems by considering that:

- the indices used for sampling with replacement are independent, so there is no reason to generate them sequentially in the master;
- the data is the same for all the slaves so we can export it explicitly just once at the beginning.

NOTE: R objects that are passed as arguments to the function executed on the slaves are automatically copied each time.

Way Less Overhead: Good Job!

```
plot(snow.time({
  snow::clusterExport(cl, c("x", "med"))
  snow::clusterEvalQ(cl, {
    indices = replicate(25, sample(10^6, 10^6, replace = TRUE),
                simplify = FALSE)
    sapply(indices, med, data = x)
  })
}))
```



Forward Model Selection (Coarse-Grained Parallelism)

Forward model selection is the most basic way to find which variables we should include in a linear regression model because they improve how well the model fits the data. If works as follows:

1. We start from a baseline, empty model (with just the intercept), and we measure how well it fits the data with BIC (lower values correspond to better models).
2. For as long as we can find better model:
   2.1 Add each candidate regressor in turn and compute BIC for the resulting model.
   2.2 If any of those models has a lower BIC than the baseline model, use that as a the new baseline model (and do not try to add that regressor again).

In a real model selection procedure, we would also try to remove terms that become redundant; we do not it here to keep the code simple. (But you are welcome to try!)

UNIVERSITY OF OXFORD

Forward Model Selection (Sequential)

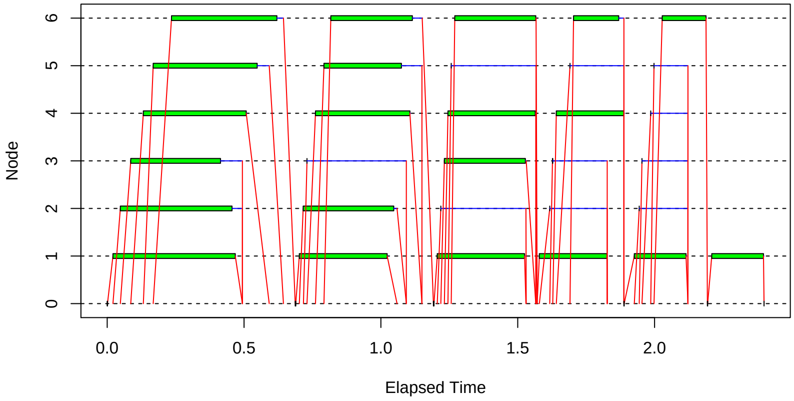
```r
linear = read.table("linear.txt", header = TRUE)
step = function(base.model, term, data)
  BIC(lm(as.formula(paste(base.model, "+", term)), data = data))

regressors = setdiff(names(linear), "F")
base.model = "F ~ 1"
base.score = BIC(lm(F ~ 1, data = linear))

repeat {

  scores = sapply(regressors, step, base.model = base.model, data = linear)

  if (min(scores) >= base.score)
    break

  base.model = paste(base.model, "+", names(which.min(scores)))
  base.score = min(scores)
  regressors = setdiff(regressors, names(which.min(scores)))

  if (length(regressors) == 0)
    break

}#REPEAT
```

Forward Model Selection (Parallel)

```
cl = makeCluster(6, type = "PSOCK")
repeat {

  scores = parSapply(cl, regressors, step,
             base.model = base.model, data = linear)

  if (min(scores) >= base.score)
    break

  base.model = paste(base.model, "+", names(which.min(scores)))
  base.score = min(scores)
  regressors = setdiff(regressors, names(which.min(scores)))

  if (length(regressors) == 0)
    break

}#REPEAT
stopCluster(cl)
```

Again, the best strategy is to base the code in one of the functions in the `apply()` family and then replace it with the corresponding function in the `parApply()` family.

Exploring Performance and Overhead

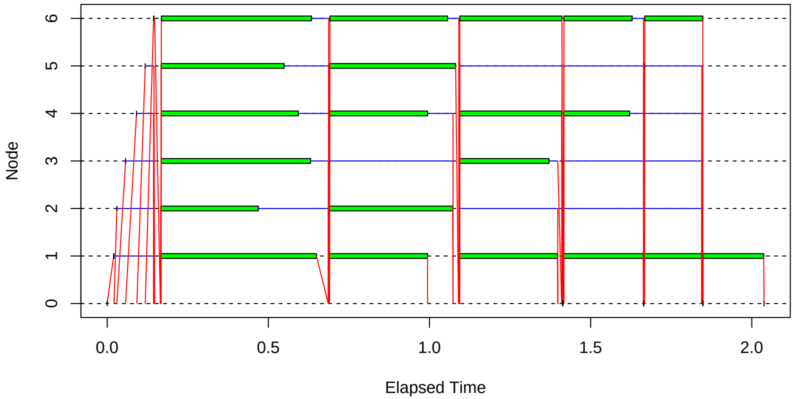


- We can clearly see we are needlessly copying data again and again; copy them in the slaves at the beginning seems worthwhile.
- We can also see the parallel parts of the algorithm interleaved with the sequential parts.

Again, With `clusterExport()`

```r
step = function(base.model, term)
  BIC(lm(as.formula(paste(base.model, "+", term)), data = linear))

clusterExport(cl, "linear")

repeat {

  scores = parSapply(cl, regressors, step,
             base.model = base.model)

  if (min(scores) >= base.score)
    break

  base.model = paste(base.model, "+", names(which.min(scores)))
  base.score = min(scores)
  regressors = setdiff(regressors, names(which.min(scores)))

  if (length(regressors) == 0)
    break

}#REPEAT
```

Sometimes The Difference is Negligible



Despite the appearances, exporting the data with `clusterExport()` does not make much difference this time.

Pro Tip: Different Functions on Different Slaves

Remember that functions are objects themselves; therefore we can pass them to the slaves as arguments, and have each slave execute different ones. Here we compute column means on one slave, and column standard deviations on the other (on the same data).

```
cl = makeCluster(2, type = "PSOCK")
slave.id = function(id)
  assign("id", value = id, envir = .GlobalEnv)
parSapply(cl, 1:2, slave.id)
```

```
## [1] 1 2
```

```
calls = list(mean, sd)
clusterExport(cl, list("calls", "linear"))
clusterEvalQ(cl, sapply(linear, calls[[id]]))
```

```
## [[1]]
##      A      B      C      D      E      F      G
##  0.998  2.028  8.009  9.025  3.505 22.051  5.011
##
## [[2]]
##    A    B    C    D    E    F    G
## 1.00 2.00 6.36 4.52 2.00 6.22 2.00
```

```
stopCluster(cl)
```

Summary and Remarks

- Many statistical methods are based on algorithms with some (or many) independent parts that can be computed at the same time: we should take advantage of that with parallel computing.
- This not something you can only do on huge compute servers, all modern desktops and laptops have at least 2-4 cores. (Be gentle to your hardware and leave one free for the operating system to use.)
- The key point is identifying how parallelisable an algorithm is; embarrassingly parallel algorithms are best.
- Consider the computational complexity (both in time and space) of your algorithm: do the parts that can run in parallel take a lot of time? Is it worthwhile?
- Investigate how much overhead you have, and try to reduce it.
- Try not to copy data around too much, and choose a sensible number of slaves based on the size of the data and the hardware.

The Hadleyverse: **dplyr** & Co.

The Problem of Base R

The **base** package in R provides a large number of options for importing, formatting and manipulating data. However, the functions that perform such tasks have been written over many years by many different people, resulting in inconsistent syntax (e.g. argument names and ordering), not-so-brilliant speed (data used to be smaller) and error messages and failure modes that are somewhat obscure. Unfortunately, these functions are used in a lot of code and therefore cannot be changed.

To address this, some developers led by Hadley Wickham developed a suite of package to provide faster and consistent replacements for those functions. We will briefly cover:

- **readr**: importing data, with useful diagnostics;
- **tidyr**: reshaping data;
- **magrittr** and **dplyr**: stringing function calls using pipes.

Importing Data with **readr**

With the classic `read.csv()`:

```
read.csv("problematic.example.csv", colClasses = c("integer", "integer"))
## Error in scan(file = file, what = what, sep = sep, quote = quote, dec =
dec, :  scan() expected 'an integer', got 'a'
```

With `read_csv()` from **readr**:

```
library(readr)
tbl = read_csv("problematic.example.csv", col_types = "ii")
problems(tbl)
## # A tibble: 2  4
##     row   col   expected actual
##   <int> <chr>      <chr>  <chr>
## 1     1     y an integer      a
## 2     2     x an integer      b
tbl[problems(tbl)$row,]
## # A tibble: 2  2
##       x     y
##   <int> <int>
## 1     1    NA
## 2    NA     2
```

Reshaping Data with **tidyr**

The **tidyr** is designed to rearrange data when

- multiple records for the same variable in the same row (e.g values at different times, one per column);
- one record spread over multiple rows (e.g. values for the same individual, one per row);
- multiple variables collated in a single column.

As an example, consider stock values for 3 stocks on 10 different days.

```
library(tidyr)

stocks = data.frame(time = as.Date('2009-01-01') + 0:9,
           X = rnorm(10, 0, 1), Y = rnorm(10, 0, 2), Z = rnorm(10, 0, 4))
head(stocks, n = 3)
```

```
##         time      X       Y      Z
## 1 2009-01-01 1.5307 -2.1723 -0.647
## 2 2009-01-02 0.9559  3.2267  7.742
## 3 2009-01-03 0.0479  0.0713  6.893
```

tidyr: Rearrange Rows and Columns

- `gather()`: one observation per row.

```
single = gather(stocks, key = stock, price, -time)
head(single, n = 3)
```

```
##         time stock  price
## 1 2009-01-01     X 1.5307
## 2 2009-01-02     X 0.9559
## 3 2009-01-03     X 0.0479
```

- `spread()`: one date or one stock per row.

```
spread(single, key = stock, price)[1:3, ]
```

```
##         time      X       Y      Z
## 1 2009-01-01 1.5307 -2.1723 -0.647
## 2 2009-01-02 0.9559  3.2267  7.742
## 3 2009-01-03 0.0479  0.0713  6.893
```

```
spread(single, key = time, price)[, 1:3]
```

```
##   stock 2009-01-01 2009-01-02
## 1     X      1.531      0.956
## 2     Y     -2.172      3.227
## 3     Z     -0.647      7.742
```

tidyr: Split and Unite Columns

- `separate()`: split the date variable into separate `year`, `month` and `day`.

```
split =
  separate(stocks, col = time, into = c("year", "month", "day"), sep = "-")
head(split, n = 5)
```

```
##   year month day       X       Y      Z
## 1 2009    01  01  1.5307 -2.1723 -0.647
## 2 2009    01  02  0.9559  3.2267  7.742
## 3 2009    01  03  0.0479  0.0713  6.893
## 4 2009    01  04 -1.1046  2.6299  1.434
## 5 2009    01  05  0.5390  1.9563  1.210
```

- `unite()`: merge `year`, `month`, `day` into a single column, separated by a `/`.

```
head(unite(split, col = date, day, month, year, sep = "/"), n = 5)
```

```
##         date       X       Y      Z
## 1 01/01/2009  1.5307 -2.1723 -0.647
## 2 02/01/2009  0.9559  3.2267  7.742
## 3 03/01/2009  0.0479  0.0713  6.893
## 4 04/01/2009 -1.1046  2.6299  1.434
## 5 05/01/2009  0.5390  1.9563  1.210
```

Unravelling Complicated Commands: **magrittr**

The most powerful aspect of R console, compared to a graphical interface, is that you can string commands together to obtain the results you need. However, this can make for long and unreadable lines of code. **magrittr** helps overcoming this problem by introducing the pipe operator %>%, which takes the return value of one function (on the left-hand side) and pass it to another function (on the right-hand side, as a first unnamed argument). So a command like this:

```
mean(sqrt(log(1:10, base=2)),trim=0.1)

## [1] 1.5
```

becomes:

```
library(magrittr)
1:10 %>%
  log(base = 2) %>%
  sqrt() %>%
  mean(trim = 0.1)

## [1] 1.5
```

magrittr and tidyr

We can combine %>% with the commands from **tidyr** to manipulate the data in multiple ways without the use of temporary variables to store intermediate results.

```
stocks %>%
  separate(col = time, into = c("year", "month", "day")) %>%
  unite(col = "day.of.year", day, month, sep = "/") %>%
  head(n = 3)
```

```
##   year day.of.year      X       Y      Z
## 1 2009       01/01 1.5307 -2.1723 -0.647
## 2 2009       02/01 0.9559  3.2267  7.742
## 3 2009       03/01 0.0479  0.0713  6.893
```

Note that:

- the pipe passes the R object to the right-hand function as its first unnamed argument, which means it is matched by position;
- naming other arguments is crucial to avoid confusion;
- not all functions take the R object as their first arguments, but you can name all previous argument in the function definition.

Slicing and Dicing the Data with **dplyr**

Finally, the **dplyr** package provides functions to complement `tidyr` in organising data before you analyse them.

```
library(dplyr)
```

There are only 5 manipulation verbs you need to express any data manipulation task; **dplyr** provides one function for each, with consistent naming: `select()`, `filter()`, `mutate()`, `summarise()` and `arrange()`. Furthermore, it provides functions such as `left_join()` and `full_join()` to merge different data frames in various ways.

NOTE: you can string these functions together with %>% and use them in combination with functions from **tidyr**; and you do not need to load `library(magrittr)` because **dplyr** does that for you.

Select Rows and Columns

- `select()`: extract or remove columns.

```
stocks %>% select(X) %>% head(n = 3)
```

```
##        X
## 1 1.5307
## 2 0.9559
## 3 0.0479
```

```
stocks %>% select(-X) %>% head(n = 3)
```

```
##         time       Y      Z
## 1 2009-01-01 -2.1723 -0.647
## 2 2009-01-02  3.2267  7.742
## 3 2009-01-03  0.0713  6.893
```

- `filter()`: choose rows.

```
stocks %>% filter(X >= Y) %>% head(n = 3)
```

```
##         time     X     Y      Z
## 1 2009-01-01  1.53 -2.17 -0.647
## 2 2009-01-09 -1.19 -1.63 -3.797
```

Join and Summarise

- `group_by()` and `summarise()`: compute per-group summary statistics.

```
stocks %>% group_by(time) %>%
  summarise(mean = mean(c(X, Y, Z)), sd = sd(c(X, Y, Z))) %>% head(n = 3)
```

```
## # A tibble: 3  3
##         time   mean    sd
##       <date>  <dbl> <dbl>
## 1 2009-01-01 -0.429  1.86
## 2 2009-01-02  3.975  3.45
## 3 2009-01-03  2.337  3.95
```

- `left_join()`: merge two different data frames sharing a common variable.

```
weekdays = data.frame(time = as.Date('2009-01-01') + 0:9,
  name = c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
           "Saturday", "Sunday", "Monday", "Tuesday", "Wednesday"))
left_join(stocks, weekdays) %>% head(n = 3)
```

```
## Joining, by = "time"
```

```
##         time      X       Y      Z      name
## 1 2009-01-01 1.5307 -2.1723 -0.647    Monday
## 2 2009-01-02 0.9559  3.2267  7.742   Tuesday
## 3 2009-01-03 0.0479  0.0713  6.893 Wednesday
```

Literate Programming and Reproducible Research

Literate Programming and Reproducible Research

The key idea behind literate programming is to mix the source code and documentation together, so that we can:

- extract the source code out (called tangle);
- execute the code to get the compiled results (called weave);
- easily update the documentation when we update the code, and vice versa, to keep them in sync;
- easily produce reports that reflects the latest results from the code.

The two main reasons to engage in literate programming are:

- keep track of code and documentation/report status in a single place;
- ensuring that research is reproducible.

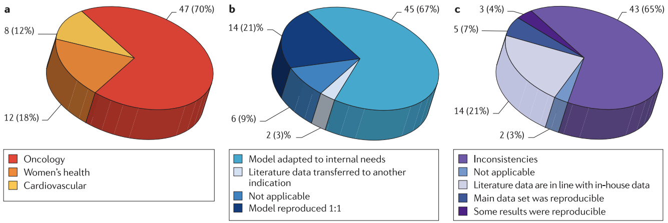
This has been achieved in R with `Sweave()`, and more recently with the **knitr** package.

Reproducibility Really is a Problem

Believe it or not: how much can we rely on published data on potential drug targets?

Florian Prinz, Thomas Schlange and Khusru Asadullah

Figure 1 | **Analysis of the reproducibility of published data in 67 in-house projects.**



Reality check on reproducibility

A survey of Nature *readers revealed a high level of concern about the problem of irreproducible results. Researchers, funders and journals need to work together to make research more reliable.*

26 MAY 2016 | VOL 533 | NATURE | 437

(And you can easily find similar papers about economics, psychology, etc.)

Good Practices

Good Practices

- Manage all files in the same directory and use relative paths.
- Work in a clean R session.
- Do not work interactively: organise your code in a collection of R functions and R scripts.
- Document which versions of R and of the R packages you are using, e.g. with `sessionInfo()`.
- If possible, keep all files in a version control system such as Git.

Common Problems

- The data is huge, so it is not trivial to archive it with the code.
- Performing exploratory analysis in batch mode is more time consuming; it is almost done interactively at least at first.
- Either the data or the code are confidential, so they cannot be stored together.

Using **knitr** with LaTeX

knitr can be used to include your R code into a LaTeX document; simply put both your LaTeX code and your R code together in a file with extension `Rnw`. The R code should be divided in chunks enclosed between `<<>>=` and `@` and placed near the text describing it.

```
Here we show a summary of an \textit{empirical} sample from a $N(2, 1)$.
<<code-chunk-label>>=
rand = rnorm(10, mean = 2, sd = 1)
summary(rand)
@
```

You can produce a LaTeX file from the `Rnw` with:

```
library(knitr)
knit("simple.Rnw")
```

or you can extract all the R code in a separate file with:

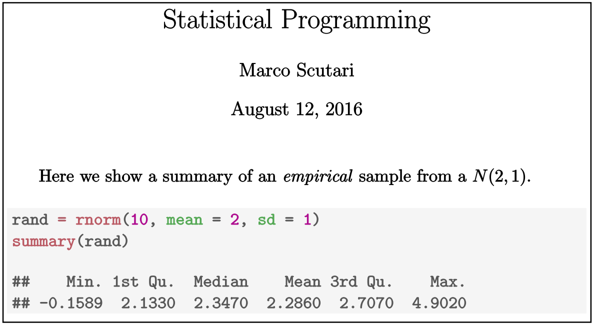
```
purl("simple.Rnw")
```

The Compiled Documents

The R file from `purl()`:

```
## ----------------------------------------------------------------------
rand = rnorm(10, mean = 2, sd = 1)
summary(rand)
```

The PDF file compiled from the LaTeX file from `knit()`:



Statistical Programming

Marco Scutari

August 12, 2016

Here we show a summary of an *empirical* sample from a $N(2, 1)$.

```
rand = rnorm(10, mean = 2, sd = 1)
summary(rand)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -0.1589  2.1330  2.3470  2.2860  2.7070  4.9020
```

Can You Customise **knitr**?

knitr provides many options that can be set either globally (for all chunks of R code) or locally (for individual chunks). The most important are:

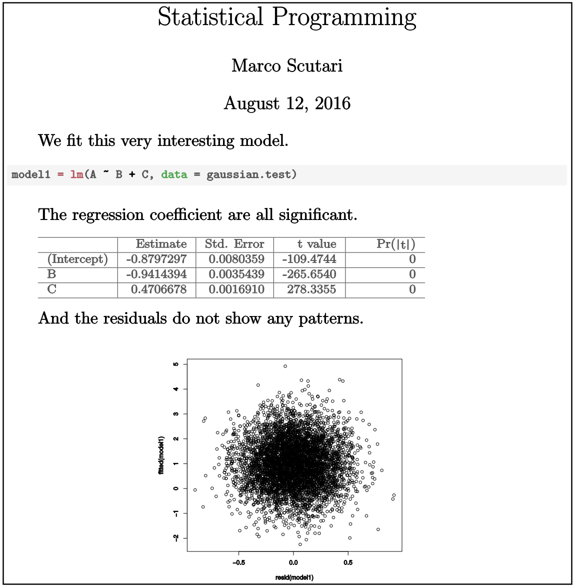
- `cache = TRUE`, `autodep = TRUE`: cache the output of R code, and run it again only when that chunk of R code changes (instead of running it every time `knit` is called). Dependencies between different chunks of code are handled automatically. Useful for long-running computations.
- `tidy = TRUE`: reorganise code to look pretty (indentation, spacing, etc.) in the output.
- `eval = FALSE`: keep the R code but not execute it. Useful for including R code in an appendix.
- `echo = FALSE`: execute the R code but do not include it in the LaTeX file. Useful for the code used to generate figures; often we want to include the figure but we do not care about the corresponding R code.

You should also be aware that **knitr** supports document formats other than LaTeX (notably HTML), and that it can generate both figures and tables with reasonably good formatting.

A Second Example: Source

```
<<formatting, echo = FALSE, cache = FALSE>>=
library(knitr)
opts_chunk$set(autodep = TRUE, cache = TRUE, tidy = FALSE,
  size = "scriptsize", keep.blank.line = FALSE,
  out.width = '6cm', out.height='6cm', fig.align = "center")
@

We fit this very interesting model.
<<model>>=
model1 = lm(A ~ B + C, data = gaussian.test)
@

The regression coefficient are all significant.

<<summary, echo = FALSE>>=
kable(summary(model1)$coefficients)
@

And the residuals do not show any patterns.
<<diagnostics, echo = FALSE>>=
plot(fitted(model1) ~ resid(model1))
@
```

A Second Example: Output



Statistical Programming

Marco Scutari

August 12, 2016

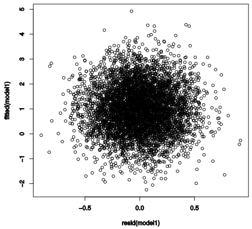
We fit this very interesting model.

```
model1 = lm(A ~ B + C, data = gaussian.test)
```

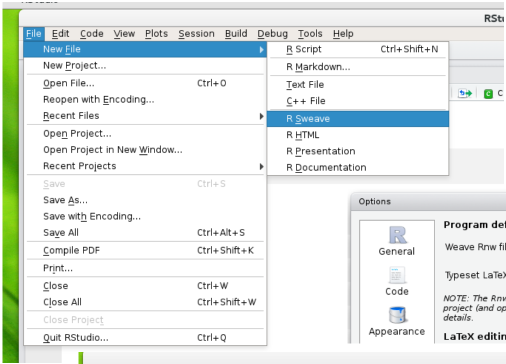
The regression coefficient are all significant.

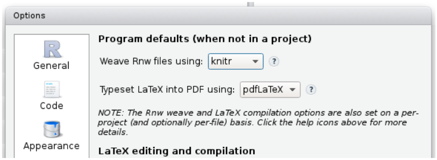
	Estimate	Std. Error	t value	Pr(|t|)
(Intercept)	-0.8797297	0.0080359	-109.4744	0
B	-0.9414394	0.0035439	-265.6540	0
C	0.4706678	0.0016910	278.3355	0

And the residuals do not show any patterns.



You Can Use **knitr** in RStudio





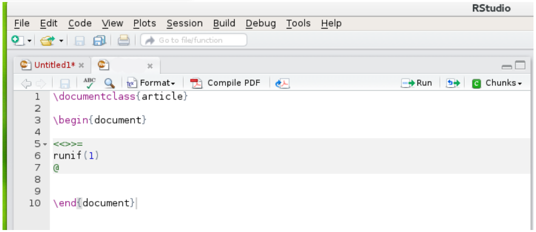
Program defaults (when not in a project)

Weave Rnw files using: knitr

Typeset LaTeX into PDF using: pdfLaTeX

NOTE: The Rnw weave and LaTeX compilation options are also set on a per-project (and optionally per-file) basis. Click the help icons above for more details.

LaTeX editing and compilation



```
\documentclass{article}

\begin{document}

<<>>=
runif(1)
@


\end{document}
```

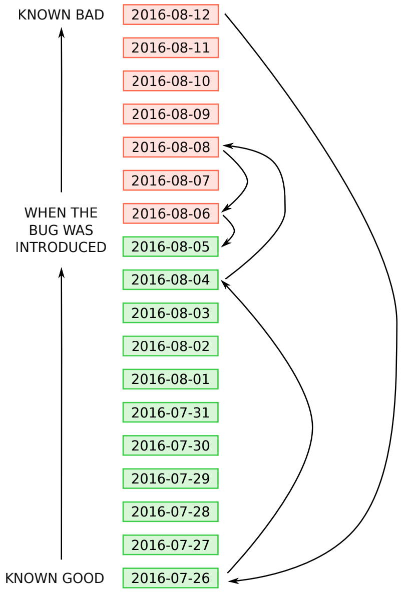
Version Control Systems

A most important feature in an editor the UNDO key - a single button that helps removing recent errors. A source code or version control system is, among other things, a giant UNDO key spanning a whole project (software, documentation, and data). It keeps track of every change you make in every file in your project so that:

- you can compare the state of your project as it was at two arbitrary points in time;
- you can work on independent changes at the same time, without those changes conflicting from each other;
- you can always rebuild the project as it existed on a given date, which is great for reproducibility.

This kind of information is invaluable for bug-tracking, performance, and quality purposes. It helps a lot when both data and R code change in the course of an analysis.

Debugging with Bisection



A debugging technique that is specific to version control systems is bisection. Given a sequence of changes to a project (here labelled by day) in which we can identify on known good state and one known bad state, a version control system makes it possible to identify which change introduced a bug by investigating all the changes in the order they were applied.

In practice, we perform a bisection search (which is optimal as it takes the minimum $O(\log_2 N)$ possible steps) until we find two consecutive states, the first without the bug and the second with the bug.

A Popular Version Control System: Git

Git is a popular version control system, and it is widely used in science and technology because:

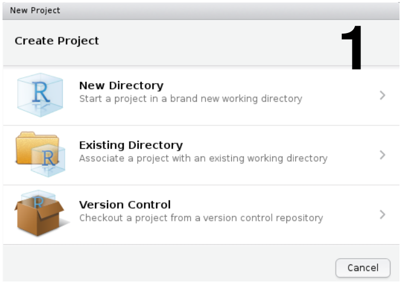
- it is free software;
- it is available for all major operating systems (Windows, Linux, OSX);
- plenty of free hosting on the web (GitHub, GitLab).

In our case, it is also important that it is integrated in RStudio.

Key steps to keep in mind:

- add one or more files to the project;
- commit the current version of the files;
- carry on your work;
- commit changes, adding a description of what has been added and what has changed;
- in case of trouble, look at the history and compare different versions of the same files to find where the problem is.

Creating a Project with Git Versioning in RStudio



New Project

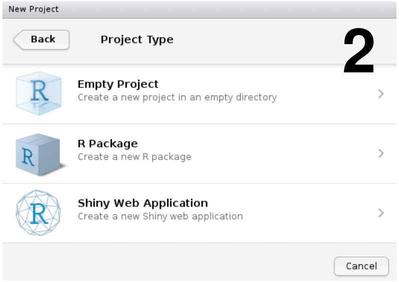
Create Project

1

New Directory
Start a project in a brand new working directory

Existing Directory
Associate a project with an existing working directory

Version Control
Checkout a project from a version control repository



New Project

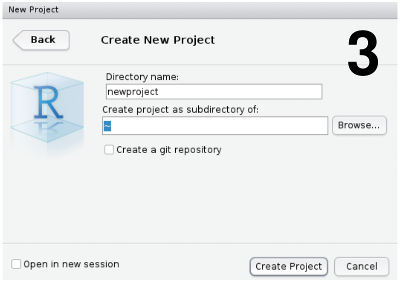
Project Type

2

Empty Project
Create a new project in an empty directory

R Package
Create a new R package

Shiny Web Application
Create a new Shiny web application



New Project

Create New Project

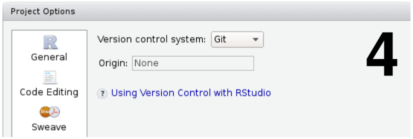
3

Directory name: newproject

Create project as subdirectory of: ~

☐ Create a git repository

☐ Open in new session



Project Options

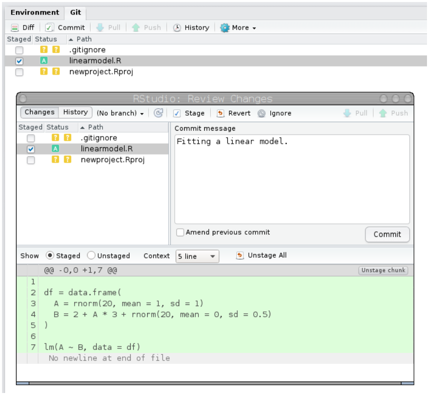
4

Version control system: Git

Origin: None

Using Version Control with RStudio

Adding A New R File, Committing Changes

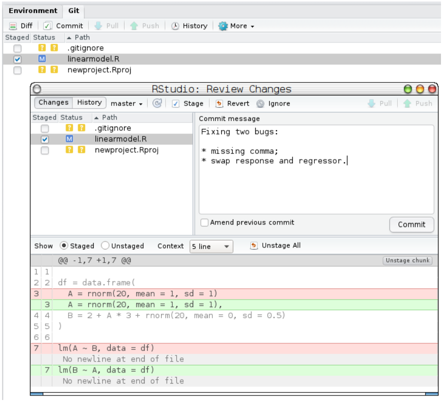


Staged	Status	Path
☐	? ?	.gitignore
✓	A	linearmodel.R
☐	? ?	newproject.Rproj

Commit message:

```
Fitting a linear model.
```

```
@@ -0,0 +1,7 @@

df = data.frame(
  A = rnorm(20, mean = 1, sd = 1)
  B = 2 + A * 3 + rnorm(20, mean = 0, sd = 0.5)
)

lm(A ~ B, data = df)
 No newline at end of file
```



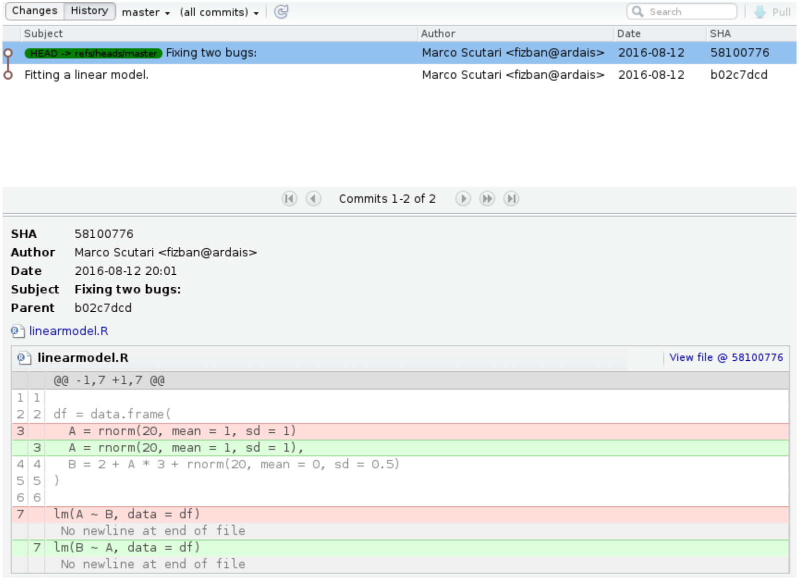
Staged	Status	Path
☐	? ?	.gitignore
✓	M	linearmodel.R
☐	? ?	newproject.Rproj

Commit message:

```
Fitting two bugs:

* missing comma;
* swap response and regressor.
```

```
@@ -1,7 +1,7 @@

df = data.frame(
  A = rnorm(20, mean = 1, sd = 1)
  A = rnorm(20, mean = 1, sd = 1),
  B = 2 + A * 3 + rnorm(20, mean = 0, sd = 0.5)
)

lm(A ~ B, data = df)
 No newline at end of file
lm(B ~ A, data = df)
 No newline at end of file
```

Visualising History



Subject	Author	Date	SHA
HEAD -> refs/heads/master Fixing two bugs:	Marco Scutari <fizban@ardais>	2016-08-12	58100776
Fitting a linear model.	Marco Scutari <fizban@ardais>	2016-08-12	b02c7dcd

Commits 1-2 of 2

SHA	58100776
Author	Marco Scutari <fizban@ardais>
Date	2016-08-12 20:01
Subject	**Fixing two bugs:**
Parent	b02c7dcd

linearmodel.R

linearmodel.R View file @ 58100776

```
      @@ -1,7 +1,7 @@
1 1
2 2   df = data.frame(
3       A = rnorm(20, mean = 1, sd = 1)
  3     A = rnorm(20, mean = 1, sd = 1),
4 4     B = 2 + A * 3 + rnorm(20, mean = 0, sd = 0.5)
5 5   )
6 6
7     lm(A ~ B, data = df)
       No newline at end of file
  7   lm(B ~ A, data = df)
       No newline at end of file
```

Summary and Remarks

- Keeping your work organised is crucial to make process smoothly and to make your results reproducible.
- Most people will not (or can not) read your R code; it is crucial to keep your documentation and your code in sync to avoid disseminating the wrong results.
- Tracking changes in your data, your code and your documentation with a version control system, and keeping it all organised in a single project is a principled way to achieve these aims.