# **Single‐cell RNA‐seq Analysis**

Single‐cell RNA ‐seq has enabled gene expression to be studied at an unprecedented resolution. The promise of this technology is attracting a growing user base for single‐cell analysis methods. As more analysis tools are becoming available, it is becoming increasingly difficult to navigate this landscape and produce an up‐to‐date workflow to analyse one's data. Here, we detail the steps of a typical single‐cell RNA ‐seq analysis, including pre‐processing (quality control, normalization, data correction, feature selection, and dimensionality reduction) and cell‐ and gene‐level downstream analysis. We formulate current best‐practice recommendations for these steps based on independent comparison studies. We have integrated these best‐practice recommendations into a workflow, which we apply to a public dataset to further illustrate how these steps work in practice. Our documented case study can be found at <https://www.github.com/theislab/single-cell-tutorial>. This review will serve as a workflow tutorial for new entrants into the field, and help established users update their analysis pipelines.

# **Portable Shiny App Based on Linux**

# How to Deploy Interactive R Apps with Shiny Server: <https://www.linode.com/docs/development/r/how-to-deploy-rshiny-server-on-ubuntu-and-debian/>

## Deploy a Shiny App to a Remote Server

### Deploy Your App

### Configure Shiny Server

# [**Deploying R shiny app as a standalone application**](https://stackoverflow.com/questions/33513544/deploying-r-shiny-app-as-a-standalone-application)**:**

a)Deploying Desktop Apps with R:[**https://www.r-bloggers.com/deploying-desktop-apps-with-r/**](https://www.r-bloggers.com/deploying-desktop-apps-with-r/)

**b)** Packaging your Shiny App as an Windows desktop app:<http://blog.analytixware.com/2014/03/packaging-your-shiny-app-as-windows.html>

c)DesktopDeployR: A framework for deploying self-contained R-based applications to the desktop -<https://github.com/wleepang/DesktopDeployR>

Check this: <https://stackoverflow.com/questions/33513544/deploying-r-shiny-app-as-a-standalone-application>

# [Develop shiny apps on linux and deploying them as desktop app on a windows machine](https://stackoverflow.com/questions/45724143/develop-shiny-apps-on-linux-and-deploying-them-as-desktop-app-on-a-windows-machi): [**https://stackoverflow.com/questions/45724143/develop-shiny-apps-on-linux-and-deploying-them-as-desktop-app-on-a-windows-machi**](https://stackoverflow.com/questions/45724143/develop-shiny-apps-on-linux-and-deploying-them-as-desktop-app-on-a-windows-machi)

### Desktop DeployR: <https://oddhypothesis.blogspot.com/2016/04/desktop-deployr.html>

No 1 is the most useful resource so far

# Optimizing R Shiny Application

1. Make Shiny fast by doing as little work as possible:<https://rstudio.com/resources/rstudioconf-2018/make-shiny-fast-by-doing-as-little-work-as-possible/>

a)Rprof and profvis

* “Feels slow” usually means R is busy
* Rprof: sample what R is doing
  + Computing (ggplot2, dplyr)
  + Waiting (database, network, disk)
* profvis: visualize Rprof output

# Optimizing Your Application:<https://bookdown.org/hadrien/how_to_build_a_shiny_app_from_scratch/optimizing-your-application.html>

a)Profvis — Interactive Visualizations for Profiling R Code:<https://rstudio.github.io/profvis/>

# [Improving shiny app loading speed](https://community.rstudio.com/t/improving-shiny-app-loading-speed/5126):<https://community.rstudio.com/t/improving-shiny-app-loading-speed/5126>

# Scaling and Performance Tuning with shinyapps.io: <https://shiny.rstudio.com/articles/scaling-and-tuning.html>

# What are the best practices to make shiny application run faster?-<https://stackoverflow.com/questions/38092398/what-are-the-best-practices-to-make-shiny-application-run-faster>

# 4 Tips to Make Your Shiny Dashboard Faster: <https://www.r-craft.org/r-news/4-tips-to-make-your-shiny-dashboard-faster/>

# The Need for Optimization:<https://engineering-shiny.org/when-optimize.html>

1. Performance:<https://mastering-shiny.org/performance.html>

First 3 examples are almost same

5-7 no point are better

# **Optimizing Shiny Code**

Shiny runs R code on the server side. So to be efficient, the R code computing your values and returning results also has to be optimized.

Optimizing R code is such a broad topic that it would be possible to write a full book about it, and in fact a lot of books about R already cover this topic. Instead of re-writing these books, we will try to point to some crucial resources you can refer to if you want to get started optimizing your R code.

* Efficient R programming (Gillespie and Lovelace [2017](https://engineering-shiny.org/optimizing-shiny-code.html#ref-colingillespie2017)), has a series of methods you can quickly put into practice for more efficient R code. (<https://englianhu.files.wordpress.com/2018/10/efficient-r-programming.pdf>)
* Advanced R (Wickham [2019b](https://engineering-shiny.org/optimizing-shiny-code.html#ref-hadleywickham2019)) has a chapter about optimizing R code (number 24). In the rest of this chapter, we will be focusing on how to optimize Shiny specifically. (<https://adv-r.hadley.nz/perf-improve.html>)
* Another one is <https://engineering-shiny.org/optimizing-shiny-code.html>

From . Efficient Programming Book (Chapter 3 ,page:80)

Many people who use R would not describe themselves as programmers. Instead, they tend to have advanced domain-level knowledge and understand standard R data structures such as vectors and data frames, but have little formal training in computing. Sound familiar? In that case, this chapter is for you. In this chapter, we will discuss “big picture” programming techniques. We cover general concepts and R programming techniques about code optimization, before describing idiomatic programming structures. We conclude the chapter by examining relatively easy ways of speeding up code using the compiler package and parallel processing using multiple CPUs.

**Top Five Tips for Efficient Programming**

1. Be careful never to grow vectors.

2. Vectorize code whenever possible.

3. Use factors when appropriate.

4. Avoid unnecessary computation by caching variables.

5. Byte compile packages for an easy performance boost.

Chapter 7. Efficient Optimization(page: 192)

**Top Five Tips for Efficient Optimization**

1. Before you start to optimize you code, ensure that you know where the bottleneck

lies; use a code profiler.

2. If the data in your data frame is all of the same type, consider converting it to a

matrix for a speed boost.

3. Use specialized row and column functions whenever possible.

4. The parallel package is ideal for Monte Carlo simulations.

5. For optimal performance, consider rewriting key parts of your code in C++.

**Code Profiling**

Often you will have working code, but simply want it to run faster. In some cases, it’s obvious

where the bottleneck lies. Sometimes you will guess, relying on intuition. A drawback of this

is that you could be wrong and waste time optimizing the wrong piece of code. To make slow

code run faster, it is important to first determine where the slow code lives. This is the

purpose of code profiling.

The Rprof() function is a built-in tool for profiling the execution of R expressions. At

regular time intervals, the profiler stops the R interpreter, records the current function call

stack, and saves the information to a file. The results from Rprof() are stochastic. Each time

we run a function R, the conditions have changed. Hence, each time you profile your code, the

result will be slightly different.

Unfortunately, Rprof() is not user-friendly. For this reason, we recommend using the profvis

package for profiling your R code. profvis provides an interactive graphical interface for

visualizing code-profiling data from Rprof().

(From Advanced R book) Chapter- 24 Improving performance

24.1 Introduction

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.

— Donald Knuth

Once you’ve used profiling to identify a bottleneck, you need to make it faster. It’s difficult to provide general advice on improving performance, but I try my best with four techniques that can be applied in many situations. I’ll also suggest a general strategy for performance optimisation that helps ensure that your faster code is still correct.

It’s easy to get caught up in trying to remove all bottlenecks. Don’t! Your time is valuable and is better spent analysing your data, not eliminating possible inefficiencies in your code. Be pragmatic: don’t spend hours of your time to save seconds of computer time. To enforce this advice, you should set a goal time for your code and optimise only up to that goal. This means you will not eliminate all bottlenecks. Some you will not get to because you’ve met your goal. Others you may need to pass over and accept either because there is no quick and easy solution or because the code is already well optimised and no significant improvement is possible. Accept these possibilities and move on to the next candidate.

If you’d like to learn more about the performance characteristics of the R language, I’d highly recommend Evaluating the Design of the R Language (Morandat et al. 2012). It draws conclusions by combining a modified R interpreter with a wide set of code found in the wild.

**Outline**

Section 24.2 teaches you how to organise your code to make optimisation as easy, and bug free, as possible.

Section 24.3 reminds you to look for existing solutions.

Section 24.4 emphasises the importance of being lazy: often the easiest way to make a function faster is to let it to do less work.

Section 24.5 concisely defines vectorisation, and shows you how to make the most of built-in functions.

Section 24.6 discusses the performance perils of copying data.

Section 24.7 pulls all the pieces together into a case study showing how to speed up repeated t-tests by about a thousand times.

Section 24.8 finishes the chapter with pointers to more resources that will help you write fast code.

**Prerequisites**

We’ll use bench to precisely compare the performance of small self-contained code chunks.

library(bench)

24.2 Code organisation

There are two traps that are easy to fall into when trying to make your code faster:

Writing faster but incorrect code.

Writing code that you think is faster, but is actually no better.

The strategy outlined below will help you avoid these pitfalls.

When tackling a bottleneck, you’re likely to come up with multiple approaches. Write a function for each approach, encapsulating all relevant behaviour. This makes it easier to check that each approach returns the correct result and to time how long it takes to run. To demonstrate the strategy, I’ll compare two approaches for computing the mean:

mean1 <- function(x) mean(x)

mean2 <- function(x) sum(x) / length(x)

I recommend that you keep a record of everything you try, even the failures. If a similar problem occurs in the future, it’ll be useful to see everything you’ve tried. To do this I recommend RMarkdown, which makes it easy to intermingle code with detailed comments and notes.

Next, generate a representative test case. The case should be big enough to capture the essence of your problem but small enough that it only takes a few seconds at most. You don’t want it to take too long because you’ll need to run the test case many times to compare approaches. On the other hand, you don’t want the case to be too small because then results might not scale up to the real problem. Here I’m going to use 100,000 numbers:

x <- runif(1e5)

Now use bench::mark() to precisely compare the variations. bench::mark() automatically checks that all calls return the same values. This doesn’t guarantee that the function behaves the same for all inputs, so in an ideal world you’ll also have unit tests to make sure you don’t accidentally change the behaviour of the function.

bench::mark(

mean1(x),

mean2(x)

)[c("expression", "min", "median", "itr/sec", "n\_gc")]

#> # A tibble: 2 x 4

#> expression min median `itr/sec`

#> <bch:expr> <bch:tm> <bch:tm> <dbl>

#> 1 mean1(x) 184µs 187.4µs 5281.

#> 2 mean2(x) 91.1µs 92.5µs 10512.

(You might be surprised by the results: mean(x) is considerably slower than sum(x) / length(x). This is because, among other reasons, mean(x) makes two passes over the vector to be more numerically accurate.)

If you’d like to see this strategy in action, I’ve used it a few times on stackoverflow:

http://stackoverflow.com/questions/22515525#22518603

http://stackoverflow.com/questions/22515175#22515856

http://stackoverflow.com/questions/3476015#22511936

24.3 Checking for existing solutions

Once you’ve organised your code and captured all the variations you can think of, it’s natural to see what others have done. You are part of a large community, and it’s quite possible that someone has already tackled the same problem. Two good places to start are:

CRAN task views. If there’s a CRAN task view related to your problem domain, it’s worth looking at the packages listed there.

Reverse dependencies of Rcpp, as listed on its CRAN page. Since these packages use C++, they’re likely to be fast.

Otherwise, the challenge is describing your bottleneck in a way that helps you find related problems and solutions. Knowing the name of the problem or its synonyms will make this search much easier. But because you don’t know what it’s called, it’s hard to search for it! The best way to solve this problem is to read widely so that you can build up your own vocabulary over time. Alternatively, ask others. Talk to your colleagues and brainstorm some possible names, then search on Google and StackOverflow. It’s often helpful to restrict your search to R related pages. For Google, try rseek. For stackoverflow, restrict your search by including the R tag, [R], in your search.

Record all solutions that you find, not just those that immediately appear to be faster. Some solutions might be slower initially, but end up being faster because they’re easier to optimise. You may also be able to combine the fastest parts from different approaches. If you’ve found a solution that’s fast enough, congratulations! Otherwise, read on.

24.3.1 Exercises

What are faster alternatives to lm()? Which are specifically designed to work with larger datasets?

What package implements a version of match() that’s faster for repeated lookups? How much faster is it?

List four functions (not just those in base R) that convert a string into a date time object. What are their strengths and weaknesses?

Which packages provide the ability to compute a rolling mean?

What are the alternatives to optim()?

24.4 Doing as little as possible

The easiest way to make a function faster is to let it do less work. One way to do that is use a function tailored to a more specific type of input or output, or to a more specific problem. For example:

rowSums(), colSums(), rowMeans(), and colMeans() are faster than equivalent invocations that use apply() because they are vectorised (Section 24.5).

vapply() is faster than sapply() because it pre-specifies the output type.

If you want to see if a vector contains a single value, any(x == 10) is much faster than 10 %in% x because testing equality is simpler than testing set inclusion.

Having this knowledge at your fingertips requires knowing that alternative functions exist: you need to have a good vocabulary. Expand your vocab by regularly reading R code. Good places to read code are the R-help mailing list and StackOverflow.

Some functions coerce their inputs into a specific type. If your input is not the right type, the function has to do extra work. Instead, look for a function that works with your data as it is, or consider changing the way you store your data. The most common example of this problem is using apply() on a data frame. apply() always turns its input into a matrix. Not only is this error prone (because a data frame is more general than a matrix), it is also slower.

Other functions will do less work if you give them more information about the problem. It’s always worthwhile to carefully read the documentation and experiment with different arguments. Some examples that I’ve discovered in the past include:

read.csv(): specify known column types with colClasses. (Also consider switching to readr::read\_csv() or data.table::fread() which are considerably faster than read.csv().)

factor(): specify known levels with levels.

cut(): don’t generate labels with labels = FALSE if you don’t need them, or, even better, use findInterval() as mentioned in the “see also” section of the documentation.

unlist(x, use.names = FALSE) is much faster than unlist(x).

interaction(): if you only need combinations that exist in the data, use drop = TRUE.

Below, I explore how you might improve apply this strategy to improve the performance of mean() and as.data.frame().

24.4.1 mean()

Sometimes you can make a function faster by avoiding method dispatch. If you’re calling a method in a tight loop, you can avoid some of the costs by doing the method lookup only once:

For S3, you can do this by calling generic.class() instead of generic().

For S4, you can do this by using selectMethod() to find the method, saving it to a variable, and then calling that function.

For example, calling mean.default() is quite a bit faster than calling mean() for small vectors:

x <- runif(1e2)

bench::mark(

mean(x),

mean.default(x)

)[c("expression", "min", "median", "itr/sec", "n\_gc")]

#> # A tibble: 2 x 4

#> expression min median `itr/sec`

#> <bch:expr> <bch:tm> <bch:tm> <dbl>

#> 1 mean(x) 2.34µs 2.62µs 348171.

#> 2 mean.default(x) 1.2µs 1.32µs 649274.

This optimisation is a little risky. While mean.default() is almost twice as fast for 100 values, it will fail in surprising ways if x is not a numeric vector.

An even riskier optimisation is to directly call the underlying .Internal function. This is faster because it doesn’t do any input checking or handle NA’s, so you are buying speed at the cost of safety.

x <- runif(1e2)

bench::mark(

mean(x),

mean.default(x),

.Internal(mean(x))

)[c("expression", "min", "median", "itr/sec", "n\_gc")]

#> # A tibble: 3 x 4

#> expression min median `itr/sec`

#> <bch:expr> <bch:tm> <bch:tm> <dbl>

#> 1 mean(x) 2.35µs 2.77µs 318684.

#> 2 mean.default(x) 1.21µs 1.34µs 691969.

#> 3 .Internal(mean(x)) 286.03ns 311.06ns 3095613.

NB: most of these differences arise because x is small. If you increase the size the differences basically disappear, because most of the time is now spent computing the mean, not finding the underlying implementation. This is a good reminder that the size of the input matters, and you should motivate your optimisations based on realistic data.

x <- runif(1e4)

bench::mark(

mean(x),

mean.default(x),

.Internal(mean(x))

)[c("expression", "min", "median", "itr/sec", "n\_gc")]

#> # A tibble: 3 x 4

#> expression min median `itr/sec`

#> <bch:expr> <bch:tm> <bch:tm> <dbl>

#> 1 mean(x) 19.8µs 20.2µs 48327.

#> 2 mean.default(x) 18.7µs 19.2µs 51566.

#> 3 .Internal(mean(x)) 17.8µs 17.8µs 55675.

24.4.2 as.data.frame()

Knowing that you’re dealing with a specific type of input can be another way to write faster code. For example, as.data.frame() is quite slow because it coerces each element into a data frame and then rbind()s them together. If you have a named list with vectors of equal length, you can directly transform it into a data frame. In this case, if you can make strong assumptions about your input, you can write a method that’s considerably faster than the default.

quickdf <- function(l) {

class(l) <- "data.frame"

attr(l, "row.names") <- .set\_row\_names(length(l[[1]]))

l

}

l <- lapply(1:26, function(i) runif(1e3))

names(l) <- letters

bench::mark(

as.data.frame = as.data.frame(l),

quick\_df = quickdf(l)

)[c("expression", "min", "median", "itr/sec", "n\_gc")]

#> # A tibble: 2 x 4

#> expression min median `itr/sec`

#> <bch:expr> <bch:tm> <bch:tm> <dbl>

#> 1 as.data.frame 994.21µs 1.06ms 942.

#> 2 quick\_df 6.59µs 7.47µs 123101.

Again, note the trade-off. This method is fast because it’s dangerous. If you give it bad inputs, you’ll get a corrupt data frame:

quickdf(list(x = 1, y = 1:2))

#> Warning in format.data.frame(if (omit) x[seq\_len(n0), , drop = FALSE] else x, :

#> corrupt data frame: columns will be truncated or padded with NAs

#> x y

#> 1 1 1

To come up with this minimal method, I carefully read through and then rewrote the source code for as.data.frame.list() and data.frame(). I made many small changes, each time checking that I hadn’t broken existing behaviour. After several hours work, I was able to isolate the minimal code shown above. This is a very useful technique. Most base R functions are written for flexibility and functionality, not performance. Thus, rewriting for your specific need can often yield substantial improvements. To do this, you’ll need to read the source code. It can be complex and confusing, but don’t give up!

24.4.3 Exercises

What’s the difference between rowSums() and .rowSums()?

Make a faster version of chisq.test() that only computes the chi-square test statistic when the input is two numeric vectors with no missing values. You can try simplifying chisq.test() or by coding from the mathematical definition.

Can you make a faster version of table() for the case of an input of two integer vectors with no missing values? Can you use it to speed up your chi-square test?

24.5 Vectorise

If you’ve used R for any length of time, you’ve probably heard the admonishment to “vectorise your code”. But what does that actually mean? Vectorising your code is not just about avoiding for loops, although that’s often a step. Vectorising is about taking a whole-object approach to a problem, thinking about vectors, not scalars. There are two key attributes of a vectorised function:

It makes many problems simpler. Instead of having to think about the components of a vector, you only think about entire vectors.

The loops in a vectorised function are written in C instead of R. Loops in C are much faster because they have much less overhead.

Chapter 9 stressed the importance of vectorised code as a higher level abstraction. Vectorisation is also important for writing fast R code. This doesn’t mean simply using map() or lapply(). Instead, vectorisation means finding the existing R function that is implemented in C and most closely applies to your problem.

Vectorised functions that apply to many common performance bottlenecks include:

rowSums(), colSums(), rowMeans(), and colMeans(). These vectorised matrix functions will always be faster than using apply(). You can sometimes use these functions to build other vectorised functions.

rowAny <- function(x) rowSums(x) > 0

rowAll <- function(x) rowSums(x) == ncol(x)

Vectorised subsetting can lead to big improvements in speed. Remember the techniques behind lookup tables (Section 4.5.1) and matching and merging by hand (Section 4.5.2). Also remember that you can use subsetting assignment to replace multiple values in a single step. If x is a vector, matrix or data frame then x[is.na(x)] <- 0 will replace all missing values with 0.

If you’re extracting or replacing values in scattered locations in a matrix or data frame, subset with an integer matrix. See Section 4.2.3 for more details.

If you’re converting continuous values to categorical make sure you know how to use cut() and findInterval().

Be aware of vectorised functions like cumsum() and diff().

Matrix algebra is a general example of vectorisation. There loops are executed by highly tuned external libraries like BLAS. If you can figure out a way to use matrix algebra to solve your problem, you’ll often get a very fast solution. The ability to solve problems with matrix algebra is a product of experience. A good place to start is to ask people with experience in your domain.

Vectorisation has a downside: it is harder to predict how operations will scale. The following example measures how long it takes to use character subsetting to look up 1, 10, and 100 elements from a list. You might expect that looking up 10 elements would take 10 times as long as looking up 1, and that looking up 100 elements would take 10 times longer again. In fact, the following example shows that it only takes about ~10x longer to look up 100 elements than it does to look up 1. That happens because once you get to a certain size, the internal implementation switches to a strategy that has a higher set up cost, but scales better.

lookup <- setNames(as.list(sample(100, 26)), letters)

x1 <- "j"

x10 <- sample(letters, 10)

x100 <- sample(letters, 100, replace = TRUE)

bench::mark(

lookup[x1],

lookup[x10],

lookup[x100],

check = FALSE

)[c("expression", "min", "median", "itr/sec", "n\_gc")]

#> # A tibble: 3 x 4

#> expression min median `itr/sec`

#> <bch:expr> <bch:tm> <bch:tm> <dbl>

#> 1 lookup[x1] 595.12ns 998.09ns 852131.

#> 2 lookup[x10] 1.53µs 1.64µs 538679.

#> 3 lookup[x100] 3.98µs 5.82µs 180303.

Vectorisation won’t solve every problem, and rather than torturing an existing algorithm into one that uses a vectorised approach, you’re often better off writing your own vectorised function in C++. You’ll learn how to do so in Chapter 25.

24.5.1 Exercises

The density functions, e.g., dnorm(), have a common interface. Which arguments are vectorised over? What does rnorm(10, mean = 10:1) do?

Compare the speed of apply(x, 1, sum) with rowSums(x) for varying sizes of x.

How can you use crossprod() to compute a weighted sum? How much faster is it than the naive sum(x \* w)?

24.6 Avoiding copies

A pernicious source of slow R code is growing an object with a loop. Whenever you use c(), append(), cbind(), rbind(), or paste() to create a bigger object, R must first allocate space for the new object and then copy the old object to its new home. If you’re repeating this many times, like in a for loop, this can be quite expensive. You’ve entered Circle 2 of the R inferno.

You saw one example of this type of problem in Section 23.2.2, so here I’ll show a slightly more complex example of the same basic issue. We first generate some random strings, and then combine them either iteratively with a loop using collapse(), or in a single pass using paste(). Note that the performance of collapse() gets relatively worse as the number of strings grows: combining 100 strings takes almost 30 times longer than combining 10 strings.

random\_string <- function() {

paste(sample(letters, 50, replace = TRUE), collapse = "")

}

strings10 <- replicate(10, random\_string())

strings100 <- replicate(100, random\_string())

collapse <- function(xs) {

out <- ""

for (x in xs) {

out <- paste0(out, x)

}

out

}

bench::mark(

loop10 = collapse(strings10),

loop100 = collapse(strings100),

vec10 = paste(strings10, collapse = ""),

vec100 = paste(strings100, collapse = ""),

check = FALSE

)[c("expression", "min", "median", "itr/sec", "n\_gc")]

#> # A tibble: 4 x 4

#> expression min median `itr/sec`

#> <bch:expr> <bch:tm> <bch:tm> <dbl>

#> 1 loop10 22.11µs 23.93µs 40916.

#> 2 loop100 691.91µs 712.54µs 1382.

#> 3 vec10 4.96µs 5.15µs 188248.

#> 4 vec100 36.15µs 36.84µs 26788.

Modifying an object in a loop, e.g., x[i] <- y, can also create a copy, depending on the class of x. Section 2.5.1 discusses this issue in more depth and gives you some tools to determine when you’re making copies.

24.7 Case study: t-test

The following case study shows how to make t-tests faster using some of the techniques described above. It’s based on an example in Computing thousands of test statistics simultaneously in R by Holger Schwender and Tina Müller. I thoroughly recommend reading the paper in full to see the same idea applied to other tests.

Imagine we have run 1000 experiments (rows), each of which collects data on 50 individuals (columns). The first 25 individuals in each experiment are assigned to group 1 and the rest to group 2. We’ll first generate some random data to represent this problem:

m <- 1000

n <- 50

X <- matrix(rnorm(m \* n, mean = 10, sd = 3), nrow = m)

grp <- rep(1:2, each = n / 2)

For data in this form, there are two ways to use t.test(). We can either use the formula interface or provide two vectors, one for each group. Timing reveals that the formula interface is considerably slower.

system.time(

for (i in 1:m) {

t.test(X[i, ] ~ grp)$statistic

}

)

#> user system elapsed

#> 0.661 0.000 0.661

system.time(

for (i in 1:m) {

t.test(X[i, grp == 1], X[i, grp == 2])$statistic

}

)

#> user system elapsed

#> 0.126 0.000 0.126

Of course, a for loop computes, but doesn’t save the values. We can map\_dbl() (Section 9.2.1) to do that. This adds a little overhead:

compT <- function(i){

t.test(X[i, grp == 1], X[i, grp == 2])$statistic

}

system.time(t1 <- purrr::map\_dbl(1:m, compT))

#> user system elapsed

#> 0.139 0.000 0.140

How can we make this faster? First, we could try doing less work. If you look at the source code of stats:::t.test.default(), you’ll see that it does a lot more than just compute the t-statistic. It also computes the p-value and formats the output for printing. We can try to make our code faster by stripping out those pieces.

my\_t <- function(x, grp) {

t\_stat <- function(x) {

m <- mean(x)

n <- length(x)

var <- sum((x - m) ^ 2) / (n - 1)

list(m = m, n = n, var = var)

}

g1 <- t\_stat(x[grp == 1])

g2 <- t\_stat(x[grp == 2])

se\_total <- sqrt(g1$var / g1$n + g2$var / g2$n)

(g1$m - g2$m) / se\_total

}

system.time(t2 <- purrr::map\_dbl(1:m, ~ my\_t(X[.,], grp)))

#> user system elapsed

#> 0.025 0.000 0.025

stopifnot(all.equal(t1, t2))

This gives us about a six-fold speed improvement.

Now that we have a fairly simple function, we can make it faster still by vectorising it. Instead of looping over the array outside the function, we will modify t\_stat() to work with a matrix of values. Thus, mean() becomes rowMeans(), length() becomes ncol(), and sum() becomes rowSums(). The rest of the code stays the same.

rowtstat <- function(X, grp){

t\_stat <- function(X) {

m <- rowMeans(X)

n <- ncol(X)

var <- rowSums((X - m) ^ 2) / (n - 1)

list(m = m, n = n, var = var)

}

g1 <- t\_stat(X[, grp == 1])

g2 <- t\_stat(X[, grp == 2])

se\_total <- sqrt(g1$var / g1$n + g2$var / g2$n)

(g1$m - g2$m) / se\_total

}

system.time(t3 <- rowtstat(X, grp))

#> user system elapsed

#> 0.010 0.000 0.011

stopifnot(all.equal(t1, t3))

That’s much faster! It’s at least 40 times faster than our previous effort, and around 1000 times faster than where we started.

24.8 Other techniques

Being able to write fast R code is part of being a good R programmer. Beyond the specific hints in this chapter, if you want to write fast R code, you’ll need to improve your general programming skills. Some ways to do this are to:

Read R blogs to see what performance problems other people have struggled with, and how they have made their code faster.

Read other R programming books, like The Art of R Programming (Matloff 2011) or Patrick Burns’ R Inferno to learn about common traps.

Take an algorithms and data structure course to learn some well known ways of tackling certain classes of problems. I have heard good things about Princeton’s Algorithms course offered on Coursera.

Learn how to parallelise your code. Two places to start are Parallel R (McCallum and Weston 2011) and Parallel Computing for Data Science (Matloff 2015).

Read general books about optimisation like Mature optimisation (Bueno 2013) or the Pragmatic Programmer (Hunt and Thomas 1990).

You can also reach out to the community for help. StackOverflow can be a useful resource. You’ll need to put some effort into creating an easily digestible example that also captures the salient features of your problem. If your example is too complex, few people will have the time and motivation to attempt a solution. If it’s too simple, you’ll get answers that solve the toy problem but not the real problem. If you also try to answer questions on StackOverflow, you’ll quickly get a feel for what makes a good question.

References

Bueno, Carlos. 2013. Mature Optimization Handbook. http://carlos.bueno.org/optimization/.

Hunt, Andrew, and David Thomas. 1990. The Pragmatic Programmer. Addison Wesley.

Matloff, Norman. 2011. The Art of R Programming. No Starch Press.

Matloff, Norman. 2015. Parallel Computing for Data Science. Chapman & Hall/CRC. http://amzn.com/1466587016.

McCallum, Q. Ethan, and Steve Weston. 2011. Parallel R. O’Reilly. http://amzn.com/B005Z29QT4.

Morandat, Floréal, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. “Evaluating the Design of the R Language.” In European Conference on Object-Oriented Programming, 104–31. Springer. http://r.cs.purdue.edu/pub/ecoop12.pdf.