Scalability

Preparing for Production

Dr Zak Varty

Scalability and Production

When put into production code gets used more and on more data.

We will likely have to consider scalability of our methods in

- Computation time
- Memory requirements

When doing so we have to balance a trade-off between development costs and usage costs.

Example: Bayesian Inference

- MCMC originally takes ~24 hours
- Identifying and amending bottlenecks in code reduced this to ~24 minutes.

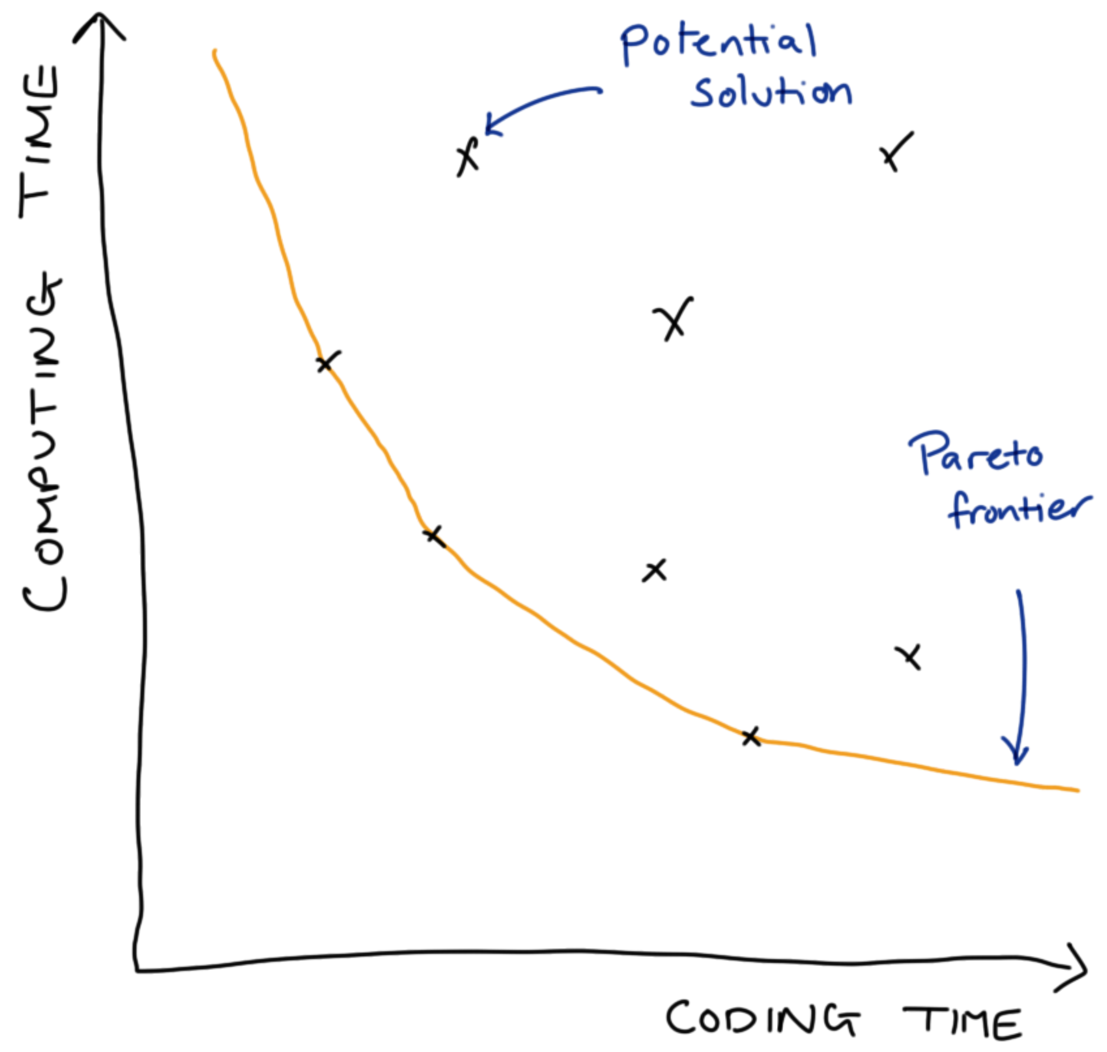
Is this actually better?

- human hours invested
- frequency of use
- safe / stable / general / readable
- trade for scalability

Knowing when to worry

Sub-optimal optimisation can be worse than doing nothing

> … programmers have spent far too much time worrying about efficiency in **the wrong places** and at **the wrong times**; premature optimisation is the root of all evil (or at least most of it) in programming. - Donald Knuth



This Lecture

- Basic profiling to find bottlenecks.
- Some simple solutions
- Strategies for scalable (R) code
- Signpost advanced methods & further reading

Profiling your code: basics

R as a stopwatch

```r
t_start <- Sys.time()
Sys.sleep(0.5) # YOUR CODE
t_end <- Sys.time()

t_end - t_start
```

```
Time difference of 0.5098979 secs
```

```r
library(tictoc)

tic()
Sys.sleep(0.5) # YOUR CODE
toc()
```

```
0.505 sec elapsed
```

With {tictoc} we can get fancy

```r
tic("total")
tic("first, easy part")
Sys.sleep(0.5)
toc(log = TRUE)
## first, easy part: 0.505 sec elapsed
tic("second, hard part")
Sys.sleep(3)
toc(log = TRUE)
## second, hard part: 3.008 sec elapsed
toc()
## total: 3.519 sec elapsed
```

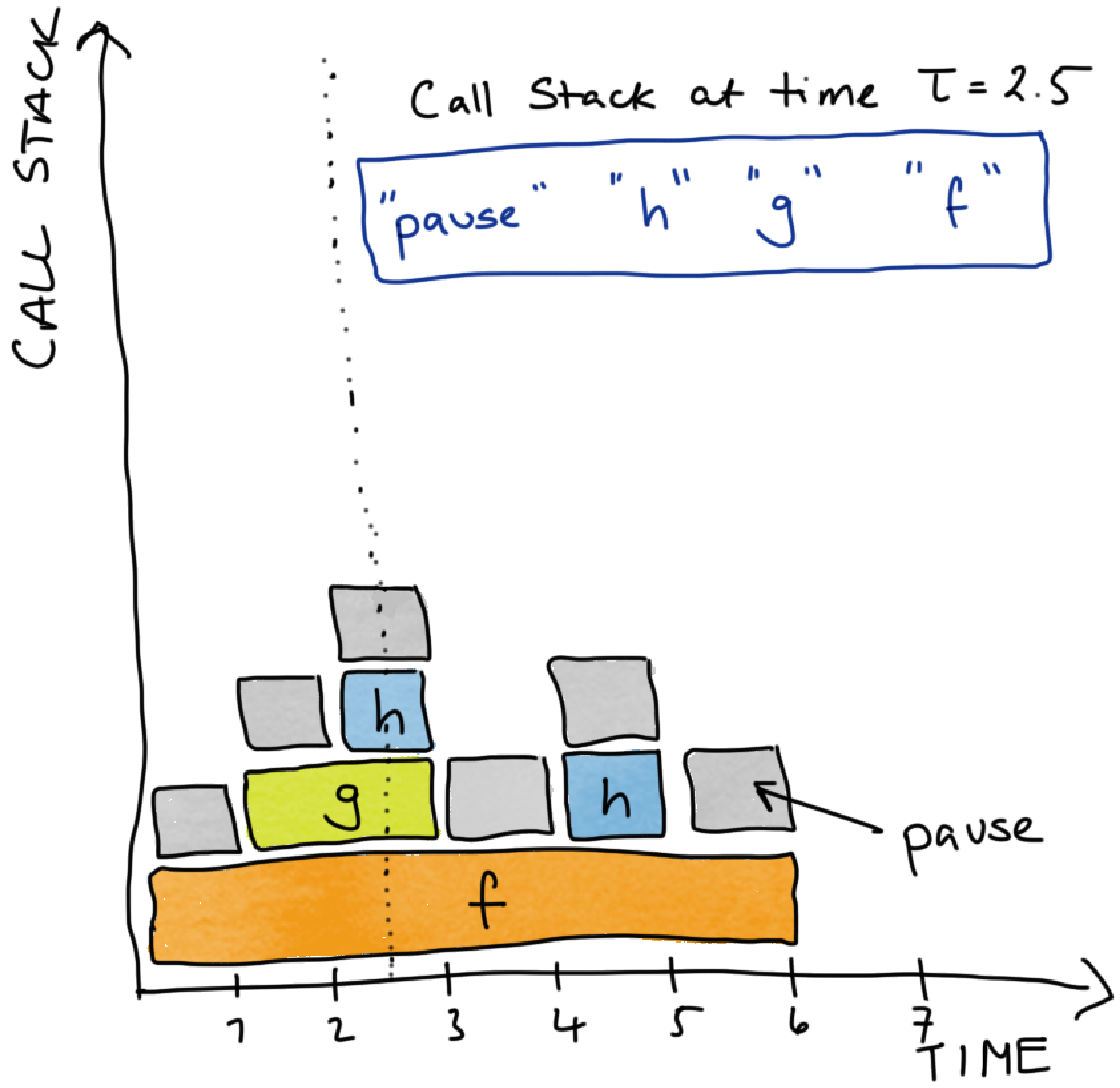
Profiling your code in detail

To diagnose scaling issues you have to understand what your code is doing.

- Stop the code at time τ and examine the **call-stack**.
  - The current function being evaluated, the function that called that, the function that called that, …, top level function.
- Do this a lot and you can measure (estimate) the proportion of working memory (RAM) uses over time and the time spent evaluating each function.

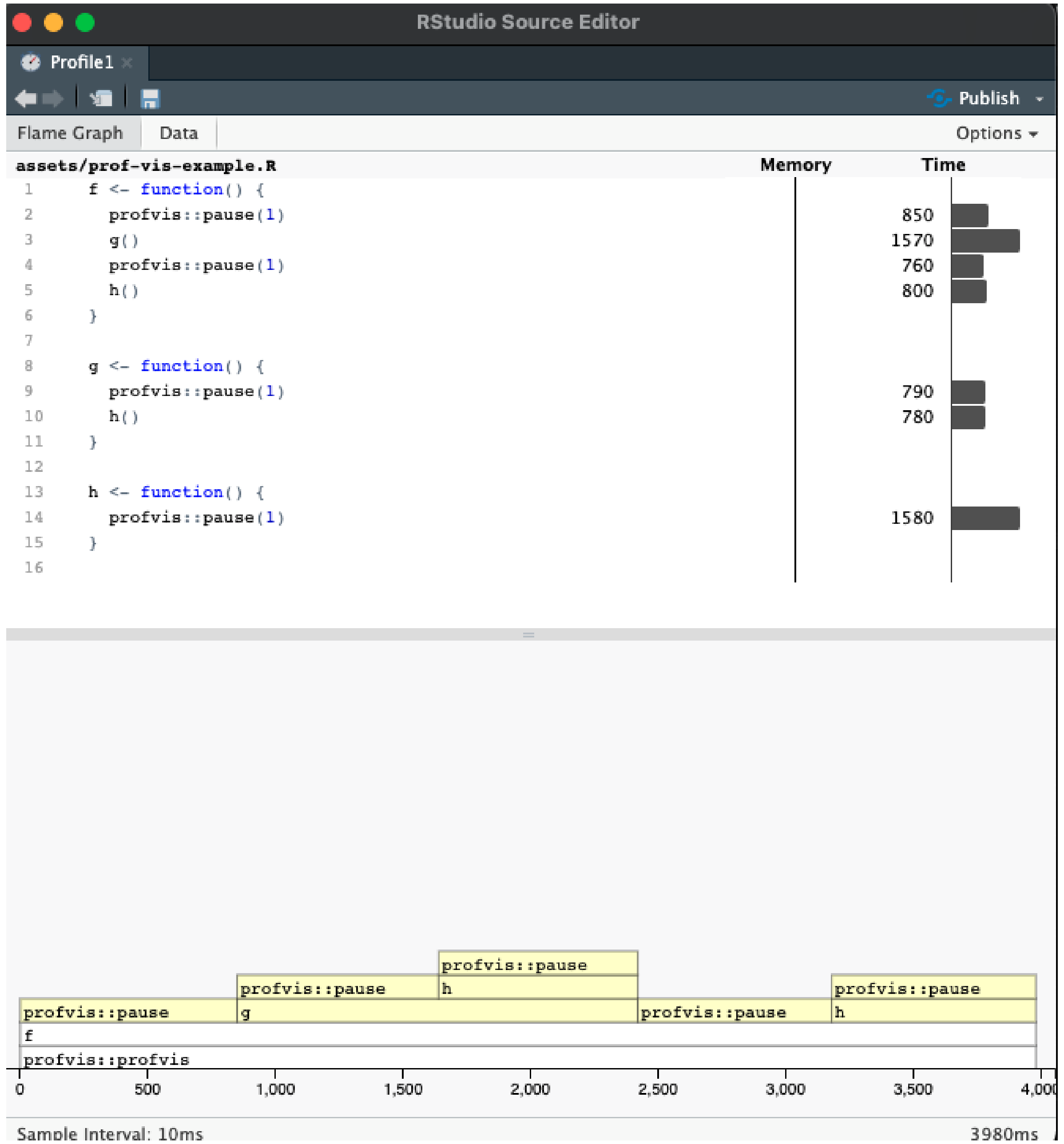
Profiling: Toy Example

```
h <- function() {
  profvis::pause(1)
}

g <- function() {
  profvis::pause(1)
  h()
}

f <- function() {
  profvis::pause(1)
  g()
  profvis::pause(1)
  h()
}
```



Profiling: How To

```r
source("assets/prof-vis-example.R")
profvis::profvis(f())
```



Notes on Time Profiling

- Will get slightly different results each time you run the function
  - Changes to internal state of computer
  - Usually not a big deal, mainly effects fastest parts of code
  - Be careful with stochastic simulations
  - Use `set.seed()` to make a fair comparison over many runs.

Notes on Profiling

Function Source

```
pad_with_NAs
```

```
function(x, n_left, n_right){
  c(rep(NA, n_left), x, rep(NA, n_right))
}
```

Compiled Function

```
mean
```

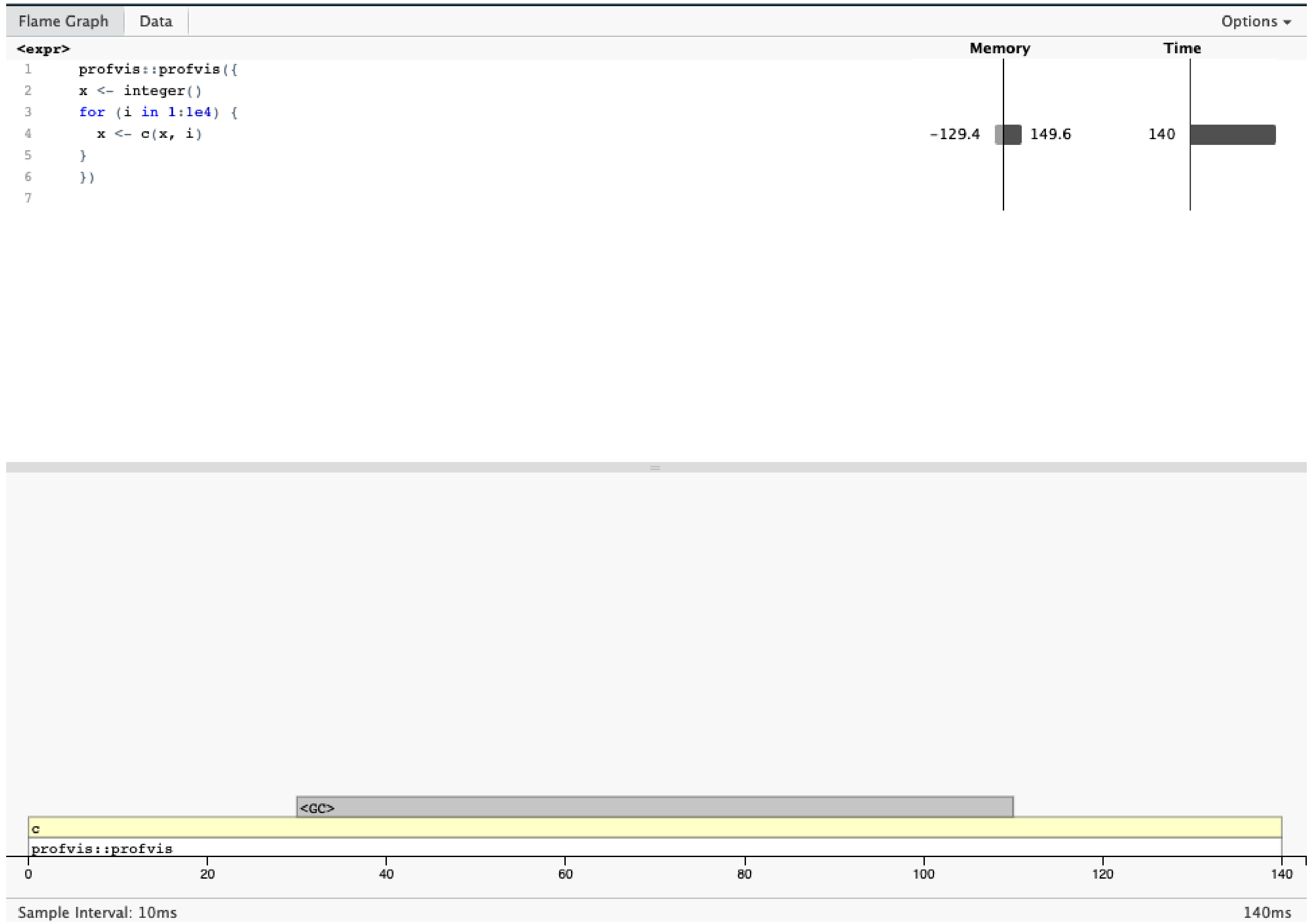
```
function (x, ...)
UseMethod("mean")
<bytecode: 0x7fabbac6d718>
<environment: namespace:base>
```

- Compiled functions have no R source code.
- Profiler does not extend into compiled code, see {jointprof} if you really need this.

Memory Profiling

`profvis()` can similarly measure the memory usage of your code.

```r
x <- integer()
for (i in 1:1e4) {
  x <- c(x, i)
}
```



- Copy-on-modify behaviour makes growing objects slow.
- Pre-allocate storage where possible.
- Strategies and structures, see R inferno and Effecient R.

Tips to work at scale

Vectorise

```r
x <- 1:10
y <- 11:20
z <- rep(NA, length(x))

for (i in seq_along(x)) {
  z[i] <- x[i] * y[i]
}
```

```r
x <- 1:10
y <- 11:20
z <- x * y
```

Use and write functions with vectorised inputs.

```r
rnorm(n = 100, mean = 1:10, sd = rep(1, 10))
```

Be careful of recycling!

Special vectors: Linear Algebra

```r
X <- diag(x = c(2, 0.5))
y <- matrix(data = c(1, 1), ncol = 1)

X %*% y
```

```
     [,1]
[1,]  2.0
[2,]  0.5
```

More on vectorising: Noam Ross Blog Post

For loops in disguise: the apply family

Functional programming equivalent of a for loop. [`apply()`, `mapply()`, `lapply()`, …]

Apply a function to each element of a list-like object.

```
A <- matrix(data = 1:12, nrow = 3, ncol = 4)
A
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```
# MARGIN = 1 => rows,  MARGIN = 2 => columns
apply(X = A, MARGIN = 1, FUN = sum)
```

```
[1] 22 26 30
```

Generalises functions from `{matrixStats}`

```
rowSums(A)
```

```
[1] 22 26 30
```

For loops in disguise: purrr::map

Iterate over a single object with `map()`:

```
mu <- c(-10, 0, 10)
purrr::map(.x = mu, .f = rnorm, n = 5)
```

```
[[1]]
[1] -10.630941 -10.536954 -10.393322  -8.395591  -8.770291

[[2]]
[1] -0.2173831  0.4013440 -2.2716535 -1.5199591  0.2817534

[[3]]
[1]  8.217122  7.209396  9.051645 10.628293 11.263494
```

Iterate over multiple objects `map2()` and `pmap()`:

```
mu <- c(-10, 0, 10)
sigma <- c(0, 0.1, 0)
purrr::map2(.x = mu, .y = sigma, .f = rnorm, n = 5)
```

```
[[1]]
[1] -10 -10 -10 -10 -10

[[2]]
[1] -0.097381610 -0.104544473 -0.063586494 -0.002777205
-0.105194011

[[3]]
[1] 10 10 10 10 10
```

For more details and variants see Advanced R chapters 9-11 on functional programming.

Easy parallelisation with furrr

- `{parallel}` and `{futures}` allow parallel coding over multiple cores.
- Powerful, but steep learning curve.
- `{furrr}` makes this very easy, just add `future_` to purrr verbs.

```
mu <- c(-10, 0, 10)
furrr::future_map(
  .x = mu,
  .f = rnorm,
  .options = furrr::furrr_options(seed = TRUE),
  n = 5)
```

```
[[1]]
[1] -11.210327  -7.806065 -10.330897  -9.628868 -11.735650

[[2]]
[1]  0.1394846 -2.5554590  0.8892910  1.0211497  1.4515885

[[3]]
[1] 10.22833  9.81417 10.17171 11.94347  9.74026
```

Need to be very careful handling RNG. See R-bloggers for more details.

Sometimes R doesn’t cut it

- An API for running C++ code in R
  - Loops that need to be run in order
  - Lots of function calls (e.g. deep recursion)
  - Fast data structures
- Beyond our scope but good to know exists. Starting point: Advanced R Chapter 25.

Wrapping up

Summary

1. Pick you battles wisely
2. Target your energy with profiling
3. Scale loops with vectors
4. Scale loops in parallel processing
5. Scale in another language

Help!

- Articles and blog links
- The R inferno (Circles 2-4)
- Advanced R (Chapters 23-25),
- Efficient R (Chapter 7).