# *Spending time to save time: simple strategies for making your code faster*

Zachary Waller

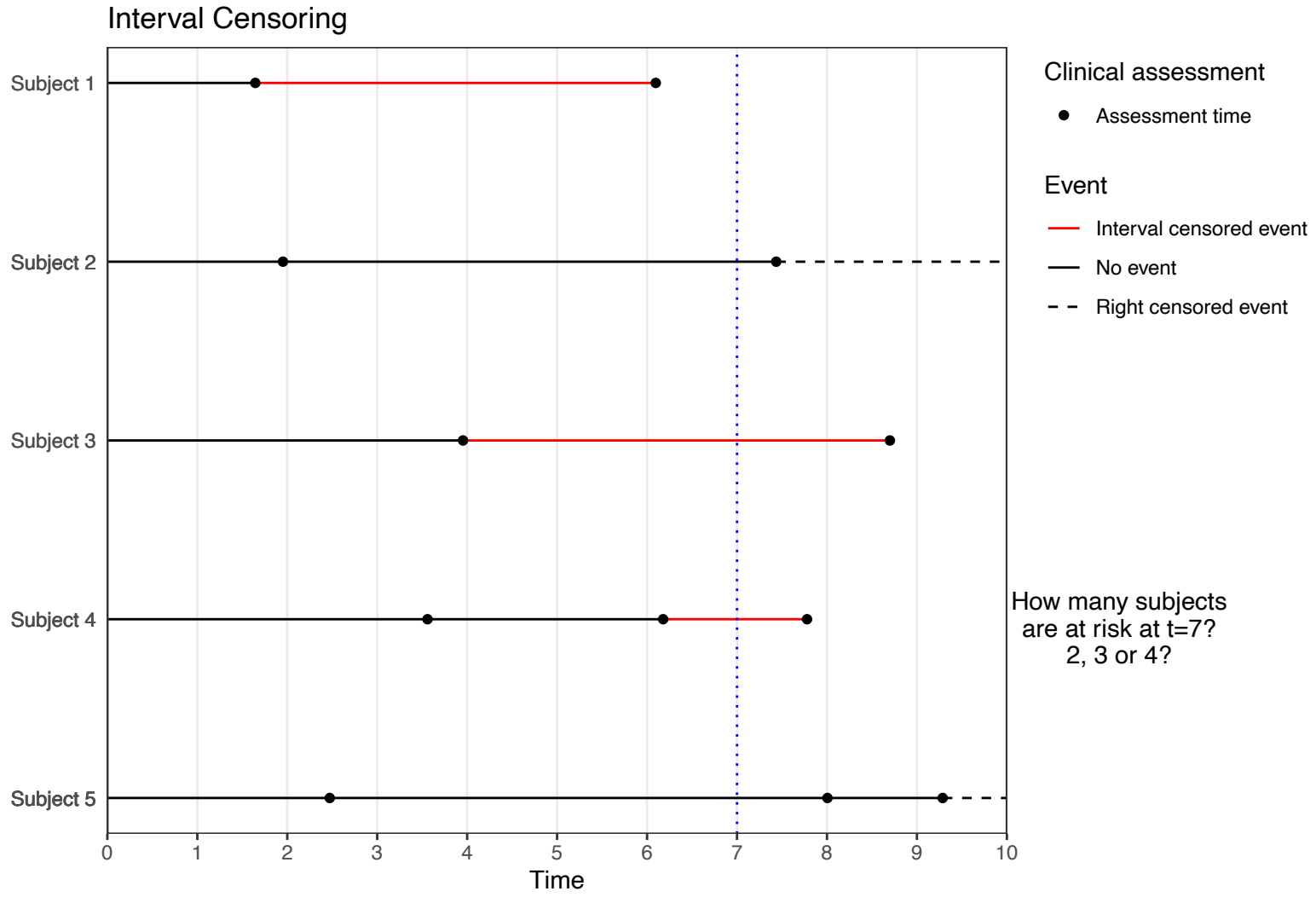
zwaller01@qub.ac.uk

QUEEN'S UNIVERSITY BELFAST

# *Motivation*

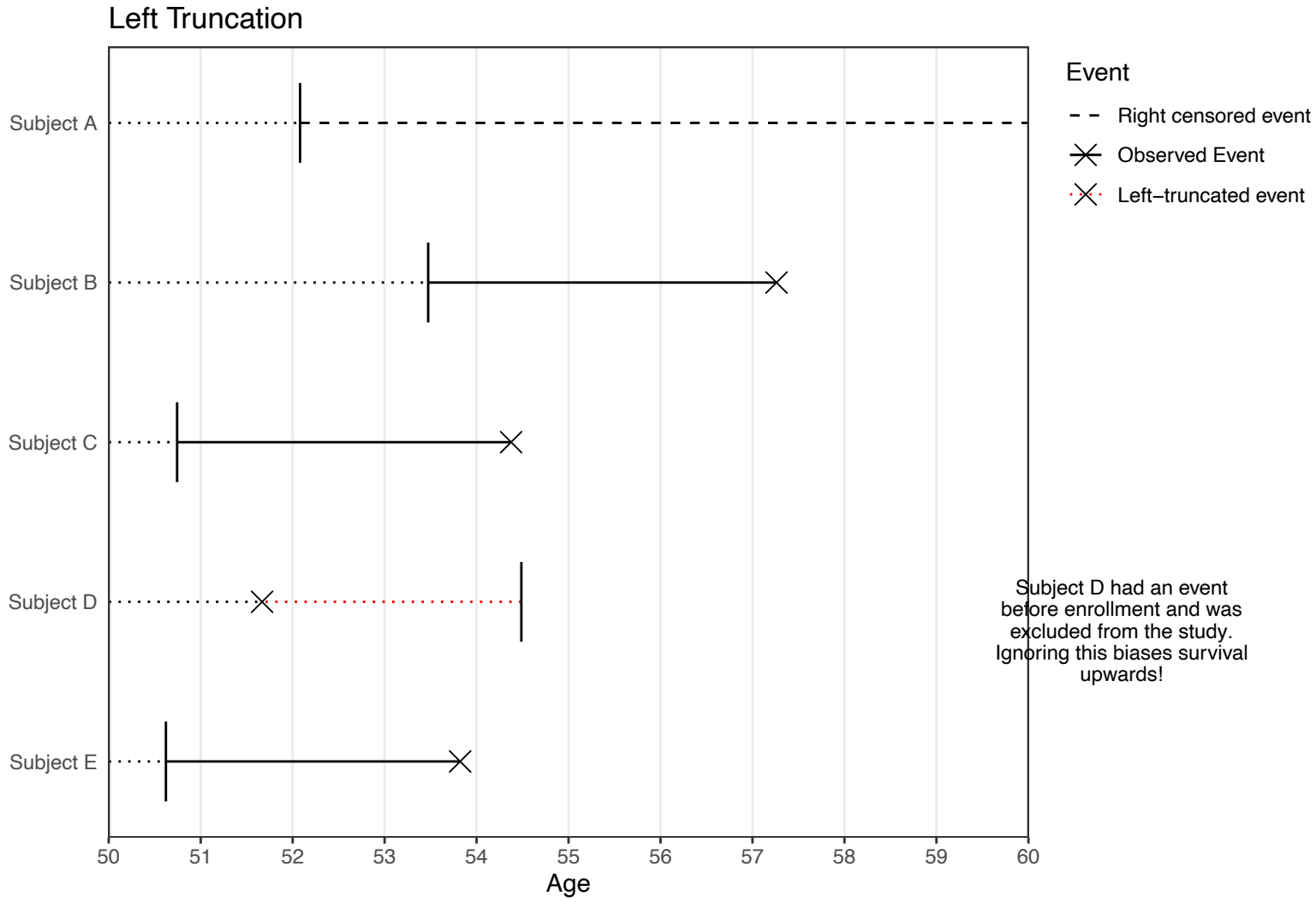
- PhD student in mathematics at Queen's University Belfast
- Developing methods for analysing survival data with interval-censored event times
  - You only know an event happened between two observation times, but not exactly when it happened
- Non-parametric maximum likelihood estimation
- No analytical solution

# Motivation




How many subjects are at risk at t=7?
2, 3 or 4?

# Motivation


Left Truncation
Subject A
Subject B
Subject C
Subject D
Subject E
50
51
52
53
54
55
56
57
58
59
60
Age
Event
Right censored event
Observed Event
Left−truncated event
Subject D had an event before enrollment and was excluded from the study. Ignoring this biases survival upwards!

# *Motivation*

- Need some iterative approach to maximum likelihood.
  - EM algorithm
  - Newton-like method
- Confidence intervals through bootstrap of likelihood-ratio
- Run your algorithm many times
- Potentially large datasets
- Speed and numeric stability are important

# *Motivation*

- Likelihood:
  - $logL = \sum_i \frac{\alpha_{ij} s_j}{\sum_j \alpha_{ij}\, s_j} - \sum_i \frac{\beta_{ij} s_j}{\sum_j \beta_{ij} s_j}$

- $i$ index over observations
- $j$ index over times

- To maximize we will have to loop through times and observations

- I chose an EM algorithm to for this problem
  - Numerical stability makes this attractive

# *How do we maximize this likelihood quickly?*

- We're interested in making the *code faster* in real terms
  - Reduce the time per iteration
  - We expect the number of iterations to stay the same (mostly)

- We're working on an iterative procedure some convergence criteria

- E.g. maximizing a likelihood, minimizing MSE, (or some other objective function)

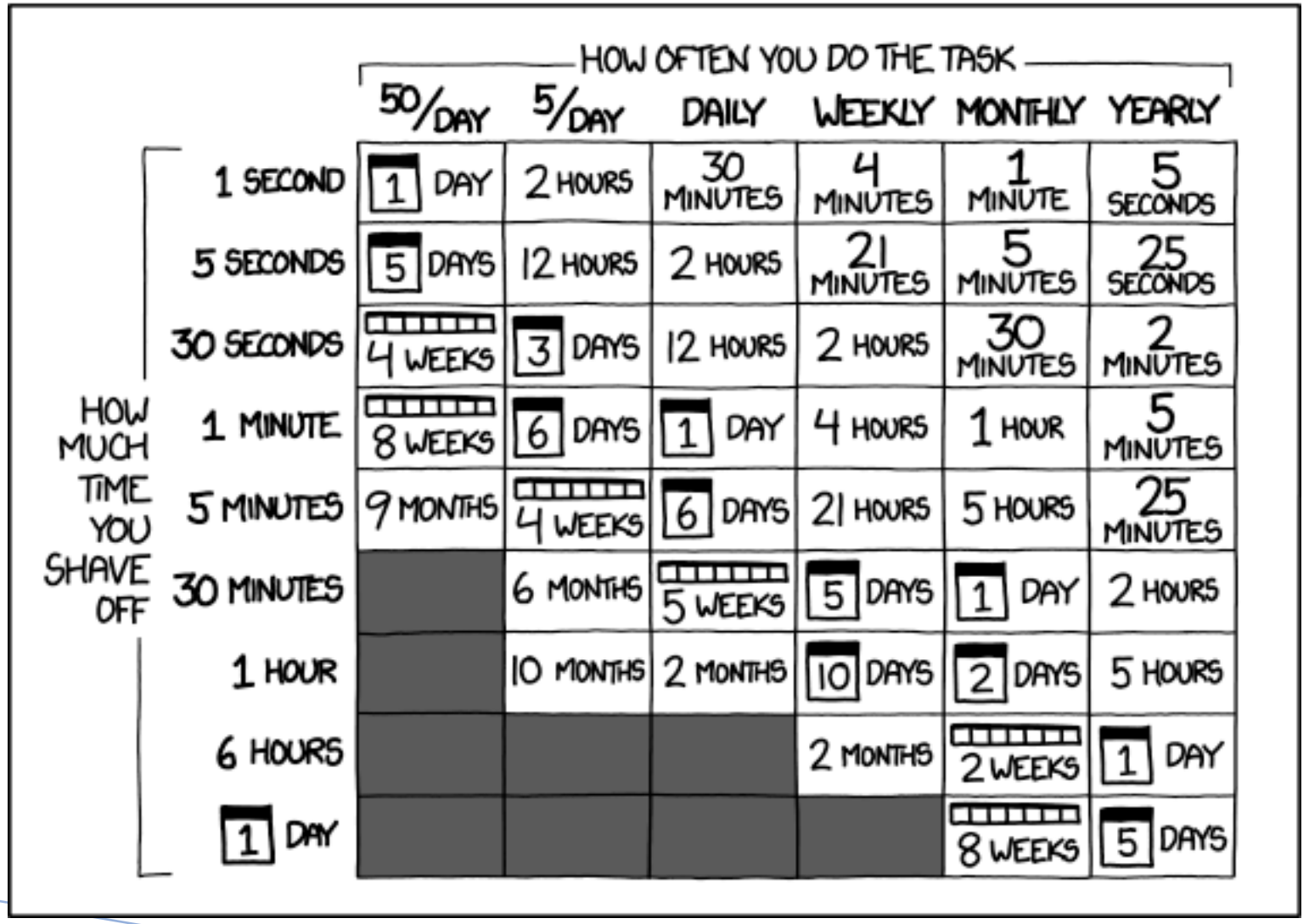
# *Some caveats*

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.
— Donald Knuth.

The real problem is that 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

# HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?

(ACROSS FIVE YEARS)

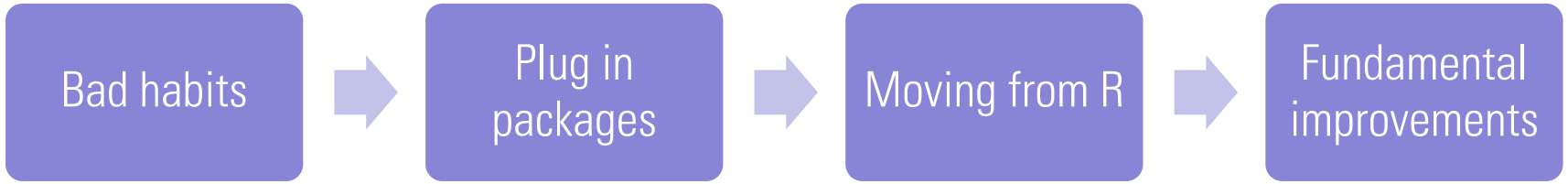



https://xkcd.com/1205

# *Easy solutions*

- Be patient
- Use a faster computer
- Use existing off the shelf optimizers
  - `optim(), nlm()`
  - Take a likelihood, gradient (optional), Hessian (optional) as input
  - Not always stable as datasets get large and subject to premature halting with left-truncation
- See if someone has solved the same problem already

# *Harder solutions*




Using some plug-in package may be the easiest way to speed up your code, but we want to avoid bad habits anyway!

# *Bad habits: not vectorizing*

- Lots of R's built in functions are vectorized by default
- This means they can take a whole vector or matrix as an argument
- Usually, some underlying C or Fortran code does the heavy lifting (i.e. they're fast).
- Being creative with these functions can speed up your code

# *Bad habits: not vectorizing*

- Not vectorized

```
for (i in 1:nrow(alpha)) {
  for (j in 1:ncol(alpha)) {
    alpha_s[[i, j]] <- alpha[[i, j]] * s[[j]]
  }
}
```

- Vectorized

```
alpha_s <- t(alpha) * s
```

Vectorized solution is around 8 times faster!

# *Vectorizing*

- For loops aren't all bad
- For loops have a reputation for always being slow in R
- The `apply()` family of functions aren't vectorized in the same way `sum()`, `colSums()` etc. are.

# *Bad habits: repetition*

- With repetition

```
t(alpha) * s) / colSums(t(alpha) * s)
```

- Without repetition

```
alpha_s <- t(alpha) * s
alpha_s / colSums(alpha_s)
```

About twice as fast when avoiding repetition!

# *Bad habits: growing objects*

- Growing every step

```r
n <- 100
my_df <- data.frame(a = character(0), b = numeric(0))

for(i in 1:n) {
  my_df <- rbind(
    my_df,
    data.frame(a = sample(letters, 1), b = runif(1)))
}
```

- Not growing every step

```r
data.frame(a = sample(letters, n, replace = TRUE), b = runif(n))
```

Over 150 times faster!

# *Bad habits: growing objects*

- Growing every step

```r
n <- 100
my_df <- data.frame(a = character(0), b = numeric(0))

for(i in 1:n) {
  my_df <- rbind(
    my_df,
    data.frame(a = sample(letters, 1), b = runif(1)))
}
```

- Not growing every step

```r
data.frame(a = sample(letters, n, replace = TRUE), b = runif(n))
```

Over 150 times faster!

# *Measuring code*

- Measuring the speed of your code is the easiest way to find bottlenecks
  - Also known as profiling

- `microbenchmark` package allows you to run small expressions to compare their speed

# *Measuring code*

```
microbenchmark(
  for_loop = {
    for (i in 1:nrow(alpha)) {
      for (j in 1:ncol(alpha)) {
        alpha_s[[i, j]] <- alpha[[i, j]] * s[[j]]
      }
    }
  },
  vectorized = {
    (t(alpha) * s)
  },
  times = 10
)
```

Here alpha is a 1000x200 matrix and s is 1x200 vector.

```
Unit: microseconds
       expr        min         lq        mean      median         uq        max neval
   for_loop 131328.084 132114.997 134573.4882 133407.1325 137500.02 138682.008    10
 vectorized    447.925    448.376    485.7885    485.0915    505.53    570.269    10
```

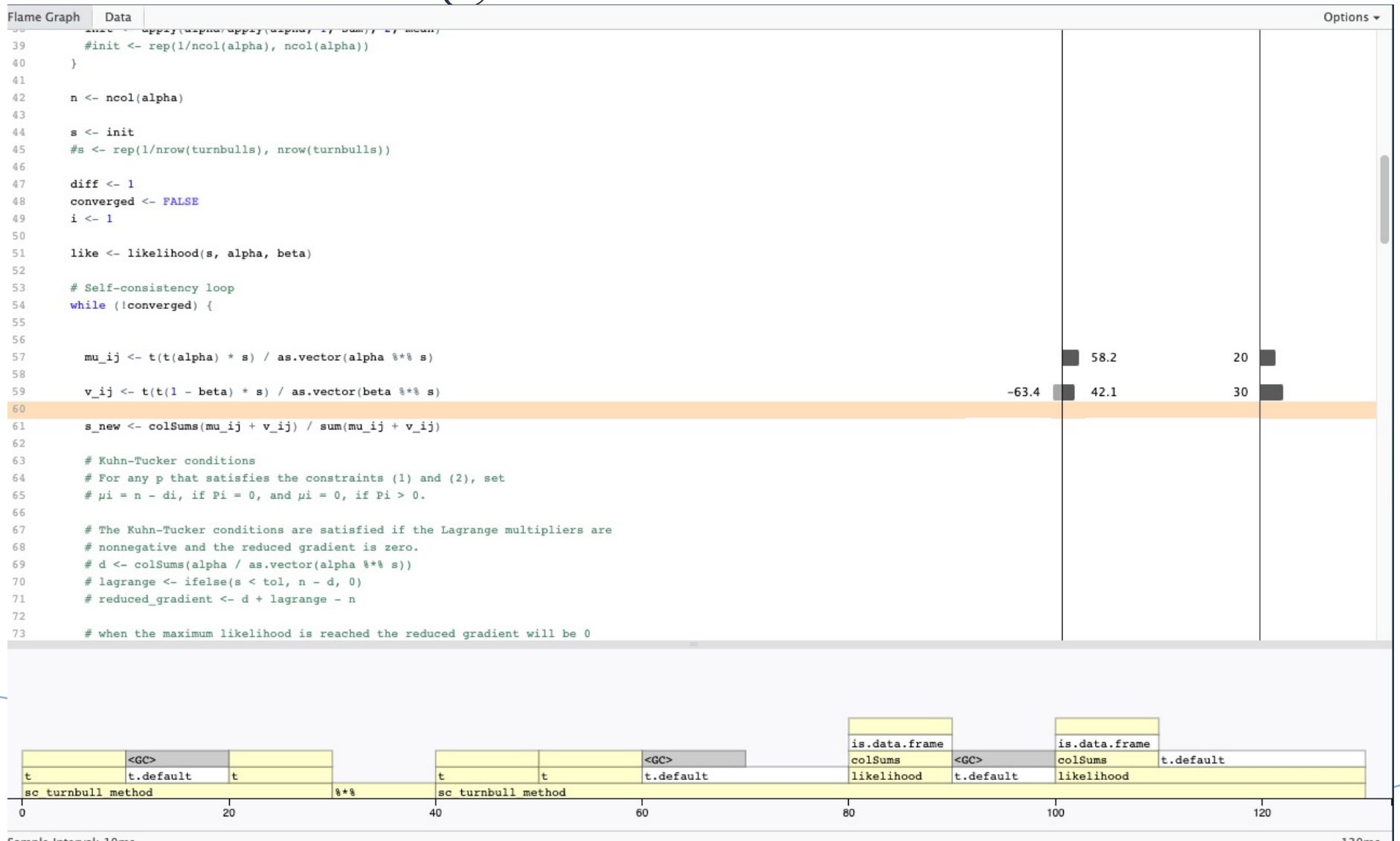
# *Measuring code*

- `profvis` package is better for larger functions and can find bottlenecks within the code

```
library(profvis)

profvis(
  {
    est <- sc_turnbull_method(left, right, max_iterations = 5000)
  }
)
```

# Measuring code



# *Parallel processing*

- Parallel processing is great when you need to run something many times
- Needs each run to be independent
- Boostrapping, sensitivity analysis, agent based models
- There is an overhead to starting the parallel process – can be slower for small problems

# *Parallel processing*

- `parallel` package

- `mclapply()` function is a parallel version of `lapply()`

- Can easily repeat functions for boostrapping data in a Monte Carlo simulation

```
boot_func <- function(i, n) {
    data <- simulate_data(n)
    sc_prodlim_method(data$lower, data$upper, data$trunc, tol = tol, max_iterations = max_it)
}

mclapply(
  1:200,
  boot_func,
  n = 100,
  mc.cores = 6
)
```

# *Problem specific solutions*

- Some packages feature off-the-shelf solutions to your problem
  - E.g. DAAREM which uses some acceleration techniques to speed up convergence of EM algorithms

```
daarem_res <- daarem::daarem(
  par = lambda,
  fixptfn = prodlim_it,
  objfn = prod_likelihood,
  alpha = event,
  beta = beta,
  y_0 = y_0,
  control = list(maxiter = 5000)
)
```

This speeds up the algorithm by actually reducing the number of iterations!

# *Experiment!*

- Sometimes it isn't obvious why a certain implementation of some code is fast and why another isn't – just messing with things can be helpful.
- If you find a bottleneck try to find a solution to that specific solution online
- Read around, talk to people and ask questions – sometimes you're missing the name of the concept that will improve your code

# *Rcpp*

- Sometimes R just isn't fast enough!
- C++ is a compiled language, which can make it very fast
- Rcpp gives a helpful interface between R and C++
- The barrier to entry is fairly high!

# *Helpful resources*

- Efficient R – Book on optimizing both your approach to coding and your code itself
- R Inferno – Book covering a number of "cardinal sins" in R programming. Slightly outdated, but a lot of great nuggets of information
- Advanced R – Book covering the ins and outs of R development with a good chapter on optimization

# *Thank you for listening*

Zachary Waller

zwaller01@qub.ac.uk

QUEEN'S UNIVERSITY BELFAST