Writing Efficient R Code

Colin Gillespie
Jumping Rivers & Newcastle University

datacamp

R version

```
version

               _
platform       x86_64-pc-linux-gnu
arch           x86_64
os             linux-gnu
system         x86_64, linux-gnu
status
major          3
minor          4.4
year           2018
month          03
day            15
svn rev        74408
language       R
version.string R version 3.4.4 (2018-03-15)
nickname       Someone to Lean On
```

Function wrapping

```
colon <- function(n) 1:n
colon(5)
```

```
1 2 3 4 5
```

```
seq_default <- function(n) seq(1, n)
seq_by <- function(n) seq(1, n, by = 1)
```

- **user time** is the CPU time charged for the execution of user instructions.
- **system time** is the CPU time charged for execution by the system on behalf of calling process.
- **elapsed time** is approximately the sum of user and system, this is the number we typically care about.

```
system.time(colon(1e8))
```

```
#   user  system elapsed
#  0.032   0.028   0.060
```

```
system.time(seq_default(1e8))
```

```
#   user  system elapsed
#  0.060   0.028   0.086
```

```
system.time(seq_by(1e8))
```

```
#   user  system elapsed
#  1.088   0.520   1.600
```

Storing the result

The trouble with

```
system.time(colon(1e8))
```

is we haven't stored the result. We need to rerun to code store the result

```
res <- colon(1e8)
```

The <- operator performs both:

- Argument passing
- Object assignment

```
system.time(res <- colon(1e8))
```

The = operator performs one of:

- Argument passing
- object assignment

```
# Raises an error
system.time(res = colon(1e8))
```

Relative time

Method	Absolute time (secs)	Relative time
`colon(n)`	0.060	0.060/0.060 = 1.00
`seq_default(n)`	0.086	0.086/0.060 = 1.40
`seq_by(n)`	1.607	1.60/0.060 = 26.7

Microbenchmark package

- Compares functions
  - Each function is run multiple times

```
library("microbenchmark")
n <- 1e8
microbenchmark(colon(n),
               seq_default(n),
               seq_by(n),
               times = 10) # Run each function 10 times
```

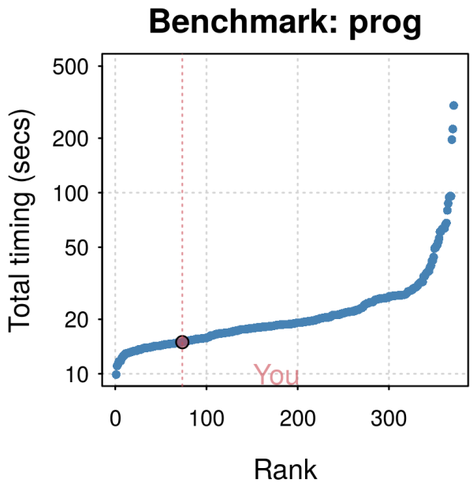
```
# Unit: milliseconds
#           expr  min   lq  mean  median   uq  max neval cld
#       colon(n)   59  130   220     202  341  391    10  a
# seq_default(n)   94  204   290     337  348  383    10  a
#      seq_by(n) 1945 2044  2260    2275 2359 2787    10   b
```

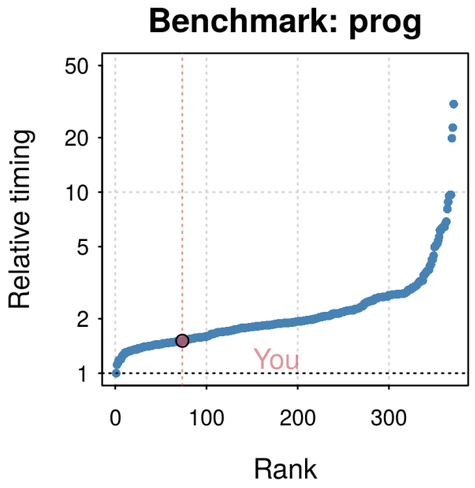
The benchmarkme package

```
install.packages("benchmarkme")
library("benchmarkme")
# Run each benchmark 3 times
res <- benchmark_std(runs = 3)
plot(res)
```

My machine is ranked 75th out 400 machines

```
upload_results(res)
```






```
# Load the benchmarkme package
library(benchmarkme)

# Assign the variable ram to the amount of RAM on this machine
ram <- get_ram()
ram

# Assign the variable cpu to the cpu specs
cpu <- get_cpu()
cpu
```

```
# Load the package
library("benchmarkme")

# Run the io benchmark. It records the length of time
# it takes to read and write a 5MB file
res <- benchmark_io(runs = 1, size = 5)

# Plot the results
plot(res)
```

Advises 1, 2, 3

- Method 1: `1:n`
- Method 2: Preallocate
- Method 3: Growing

TIME IN SECONDS

n	1	2	3
10^5	0.00	0.02	0.2
10^6	0.00	0.2	30
10^7	0.00	2	3800

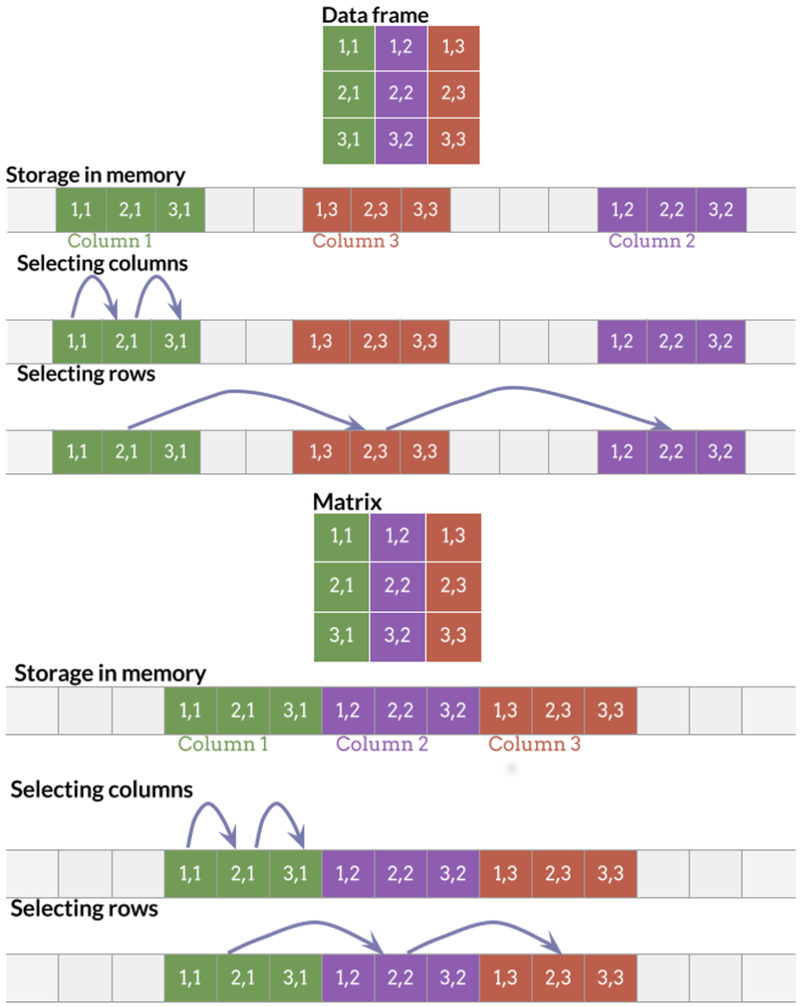
1. Never, ever grow a vector.
Pre-allocating the vector is significantly faster than growing the vector!

```
n <- 30000
# Fast code
pre_allocate <- function(n) {
    x <- numeric(n) # Pre-allocate
    for(i in 1:n)
        x[i] <- rnorm(1)
    x
}
```

2. Use a vectorized solution wherever possible.

```
x <- rnorm(n)
```

3. Use a matrix whenever appropriate.

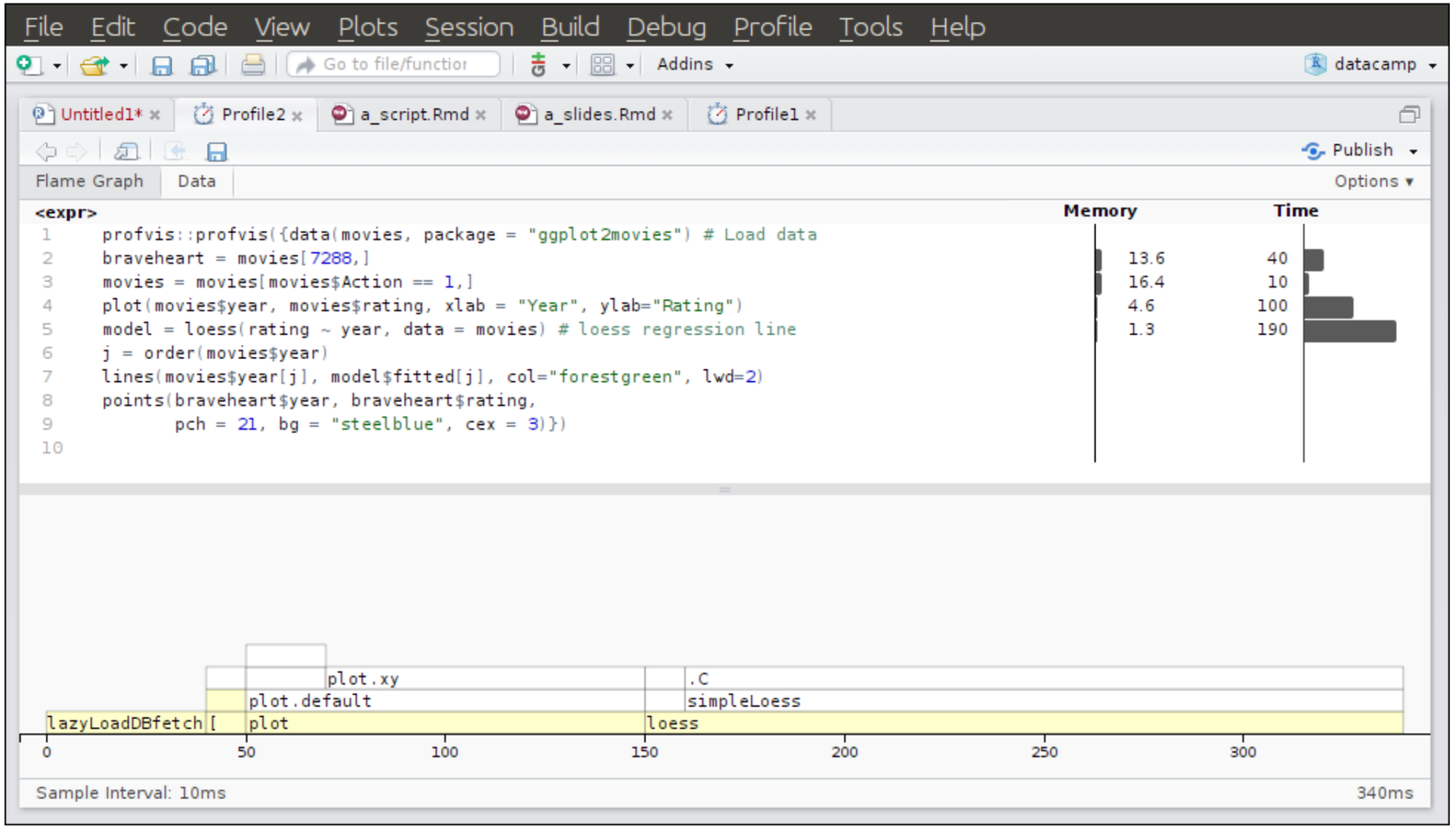


Command line

```
library("profvis")
profvis({
data(movies, package = "ggplot2movies") # Load data
braveheart <- movies[7288,]
movies <- movies[movies$Action == 1,]
plot(movies$year, movies$rating, xlab = "Year", ylab="Rating")
model <- loess(rating ~ year, data = movies) # loess regression line
j <- order(movies$year)
lines(movies$year[j], model$fitted[j], col="forestgreen", lwd=2)
points(braveheart$year, braveheart$rating,
     pch = 21, bg = "steelblue", cex = 3)
})
```

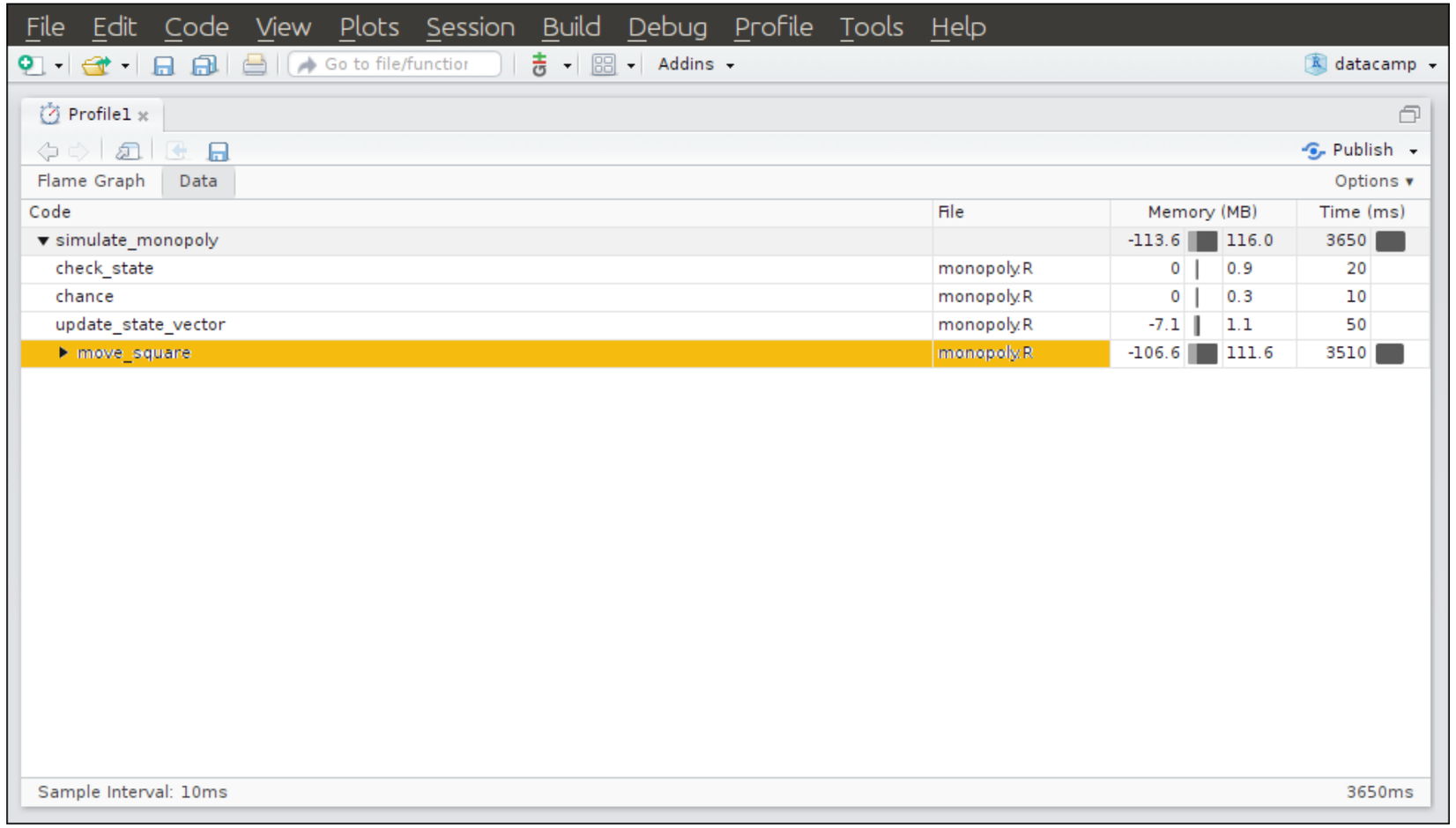
knitr::purl("path to the input file"):
This function takes an input file, extracts the R code in it according to a list of patterns, evaluates the code and writes the output in another file.

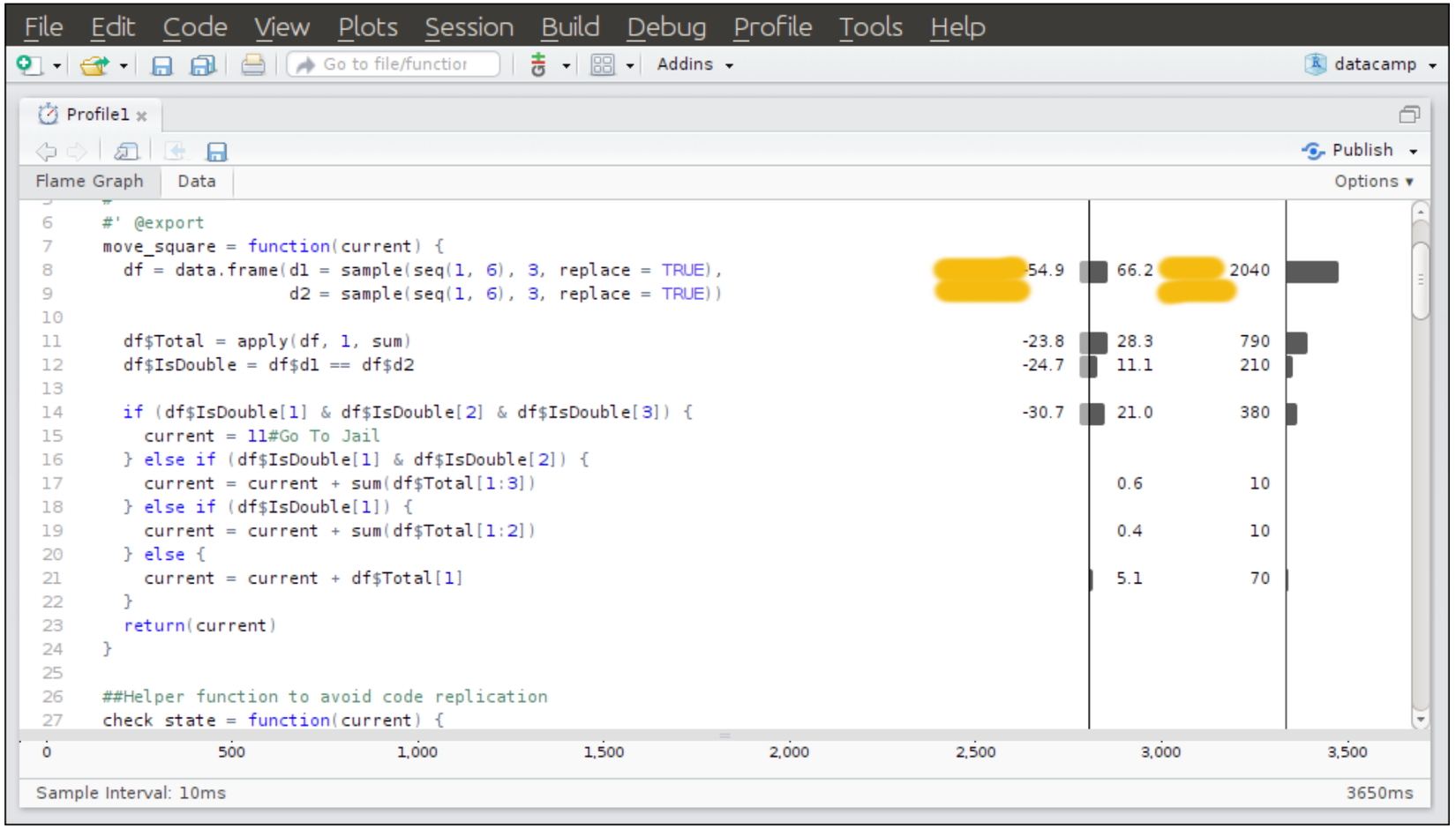
Which line do you think will be the slowest?


File Edit Code View Plots Session Build Debug Profile Tools Help
Addins
datacamp
Untitled1*
Profile2
a_script.Rmd
a_slides.Rmd
Profile1
Publish
Flame Graph
Data
Options
<expr>
Memory
Time
1 profvis::profvis({data(movies, package = "ggplot2movies") # Load data
2 braveheart = movies[7288,]
3 movies = movies[movies$Action == 1,]
4 plot(movies$year, movies$rating, xlab = "Year", ylab="Rating")
5 model = loess(rating ~ year, data = movies) # loess regression line
6 j = order(movies$year)
7 lines(movies$year[j], model$fitted[j], col="forestgreen", lwd=2)
8 points(braveheart$year, braveheart$rating,
9 pch = 21, bg = "steelblue", cex = 3)})
10
13.6
40
16.4
10
4.6
100
1.3
190
plot.xy
.C
plot.default
simpleLoess
lazyLoadDBfetch
[
plot
loess
0
50
100
150
200
250
300
Sample Interval: 10ms
340ms

Monopoly

- 40 squares
  - 28 properties (22 streets + 4 stations + 2 utilities)
    - Players take turns moving by rolling dice
  - Buying properties
  - Charging other players
    - Sent to jail: three consecutive doubles in a single turn


File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/functior
Addins
datacamp
Profile1
Publish
Flame Graph
Data
Options
Code
File
Memory (MB)
Time (ms)
simulate_monopoly
-113.6
116.0
3650
check_state
monopoly.R
0
0.9
20
chance
monopoly.R
0
0.3
10
update_state_vector
monopoly.R
-7.1
1.1
50
move_square
monopoly.R
-106.6
111.6
3510
Sample Interval: 10ms
3650ms


File Edit Code View Plots Session Build Debug Profile Tools Help
Addins
datacamp
Profile1
Publish
Flame Graph
Data
Options
#' @export
move_square = function(current) {
df = data.frame(d1 = sample(seq(1, 6), 3, replace = TRUE),
d2 = sample(seq(1, 6), 3, replace = TRUE))
df$Total = apply(df, 1, sum)
df$IsDouble = df$d1 == df$d2
if (df$IsDouble[1] & df$IsDouble[2] & df$IsDouble[3]) {
current = 11#Go To Jail
} else if (df$IsDouble[1] & df$IsDouble[2]) {
current = current + sum(df$Total[1:3])
} else if (df$IsDouble[1]) {
current = current + sum(df$Total[1:2])
} else {
current = current + df$Total[1]
}
return(current)
}
##Helper function to avoid code replication
check state = function(current) {
54.9
66.2
2040
-23.8
28.3
790
-24.7
11.1
210
-30.7
21.0
380
0.6
10
0.4
10
5.1
70
0
500
1,000
1,500
2,000
2,500
3,000
3,500
Sample Interval: 10ms
3650ms

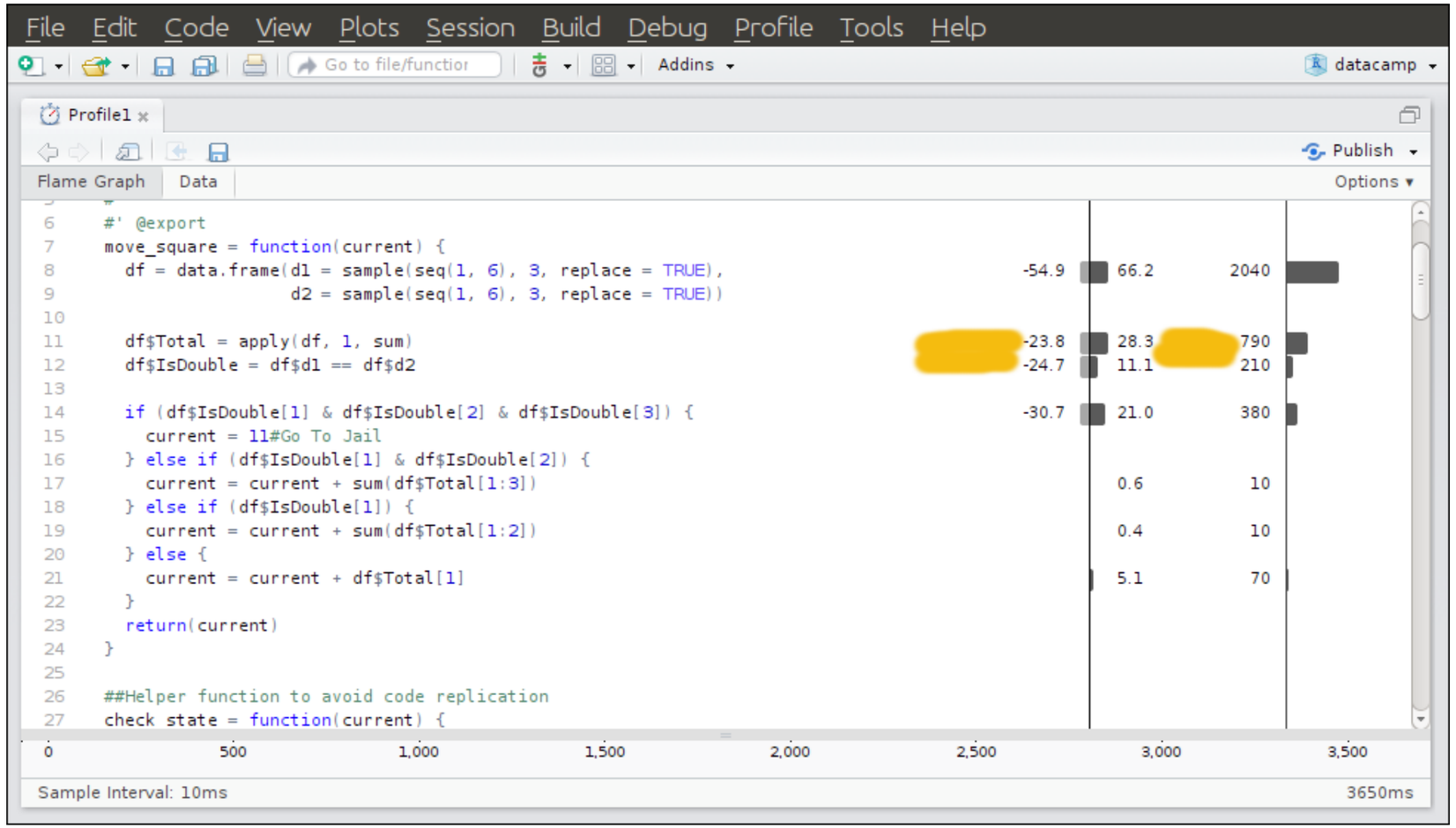
Data frames vs matrices

```
# Original
rolls <- data.frame(d1 = sample(1:6, 3, replace = TRUE),
                    d2 = sample(1:6, 3, replace = TRUE))
```

```
# Updated
rolls <- matrix(sample(1:6, 6, replace = TRUE), ncol = 2)
```

- Total Monopoly simulation time: 2 seconds to 0.5 seconds
- Creating a data frame is slower than a matrix
- In the Monopoly simulation, we created 10,000 data frames

Monopoly profvis



How would you optimize this code?

apply vs rowSums

```
# Original
total <- apply(df, 1, sum)
```

```
# Updated
total <- rowSums(df)
```

- 0.5 seconds to 0.16 seconds - 3 fold speed up

& vs &&

```
# Original
is_double[1] & is_double[2] & is_double[3]
```

```
# Updated
is_double[1] && is_double[2] && is_double[3]
```

- Limited speed-up
- 0.16 seconds to 0.15 seconds

Parallel computing

Can the loop be run forward and backwards?

```
x <- 1:8
for(i in 2:8)
    x[i] <- x[i-1]
for(i in 8:2)
    x[i] <- x[i-1]
```

- The loops give di erent answers
  - The rst: `x[8] = x[7] = ... = 1`
  - The second: `x[8] = x[7] = 7`
- Can't use parallel computing

Remember: If you can run your loop in reverse, you can probably use parallel computing.

The apply() function

- `apply()` is similar to a for loop
  - We apply a function to each row/column of a matrix
- A 10 column, 10,000 row matrix:

```
m <- matrix(rnorm(100000), ncol = 10)
```

- apply is neater than a for loop

```
res <- apply(m, 1, median)
```

Converting to parallel

```
library("parallel")
```

```
copies_of_r <- 7
```

```
cl <- makeCluster(copies_of_r)
```

```
parApply(cl, m, 1, median)
```

```
stopCluster(cl)
```

- Sometimes running in parallel is slower due to thread communication
- **Benchmark both solutions**

The apply family

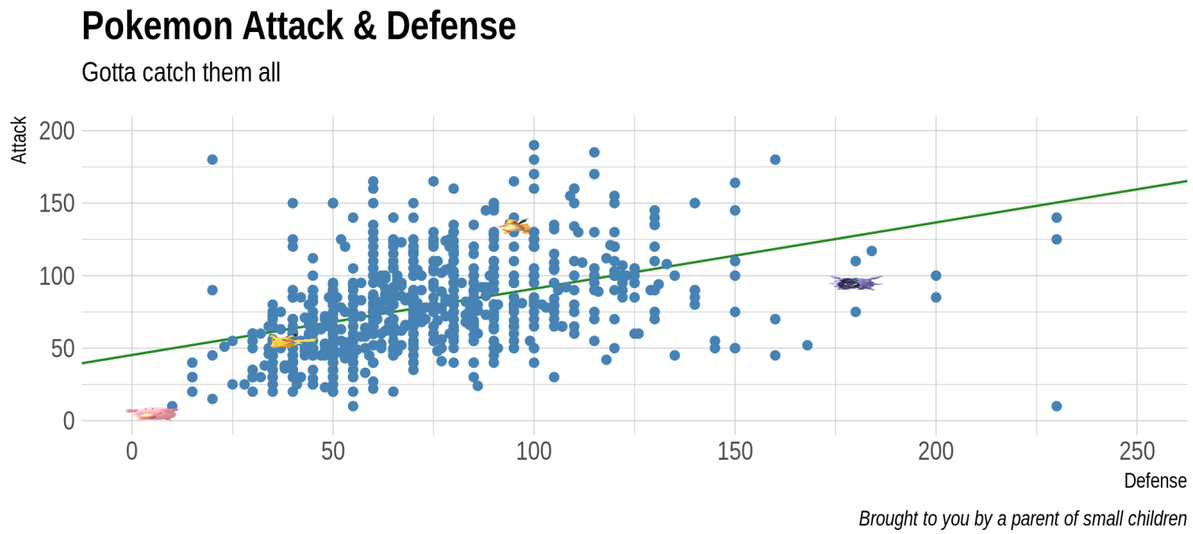
There are parallel versions of

- `apply()` - `parApply()`
- `sapply()` - `parSapply()`
  - applying a function to a vector, i.e. a for loop
- `lapply()` - `parLapply()`
  - applying a function to a list

Example: Pokemon battles

```
plot(pokemon$Defense, pokemon$Attack)
abline(lm(pokemon$Attack ~ pokemon$Defense), col = 2)
cor(pokemon$Attack, pokemon$Defense)
```

```
0.437
```



Bootstrapping

In a perfect world, we would resample from the population; but we can't

Instead, we assume the original sample is representative of the population

1. Sample with replacement from your data
   - The same point could appear multiple times
2. Calculate the correlation statistics from your new sample
3. Repeat

A single bootstrap

```
bootstrap <- function(data_set) {
    # Sample with replacement
    s <- sample(1:nrow(data_set), replace = TRUE)
    new_data <- data_set[s,]

    # Calculate the correlation
    cor(new_data$Attack, new_data$Defense)
}
```

```
# 100 independent bootstrap simulations
sapply(1:100, function(i) bootstrap(pokemon))
```

Converting to parallel

- Load the package
- Specify the number of cores
- Create a cluster object
- Export functions/data (para que el remplazo se posible)
- Swap to `parSapply()`
- Stop!

```
library("parallel")
```

```
no_of_cores <- 7
```

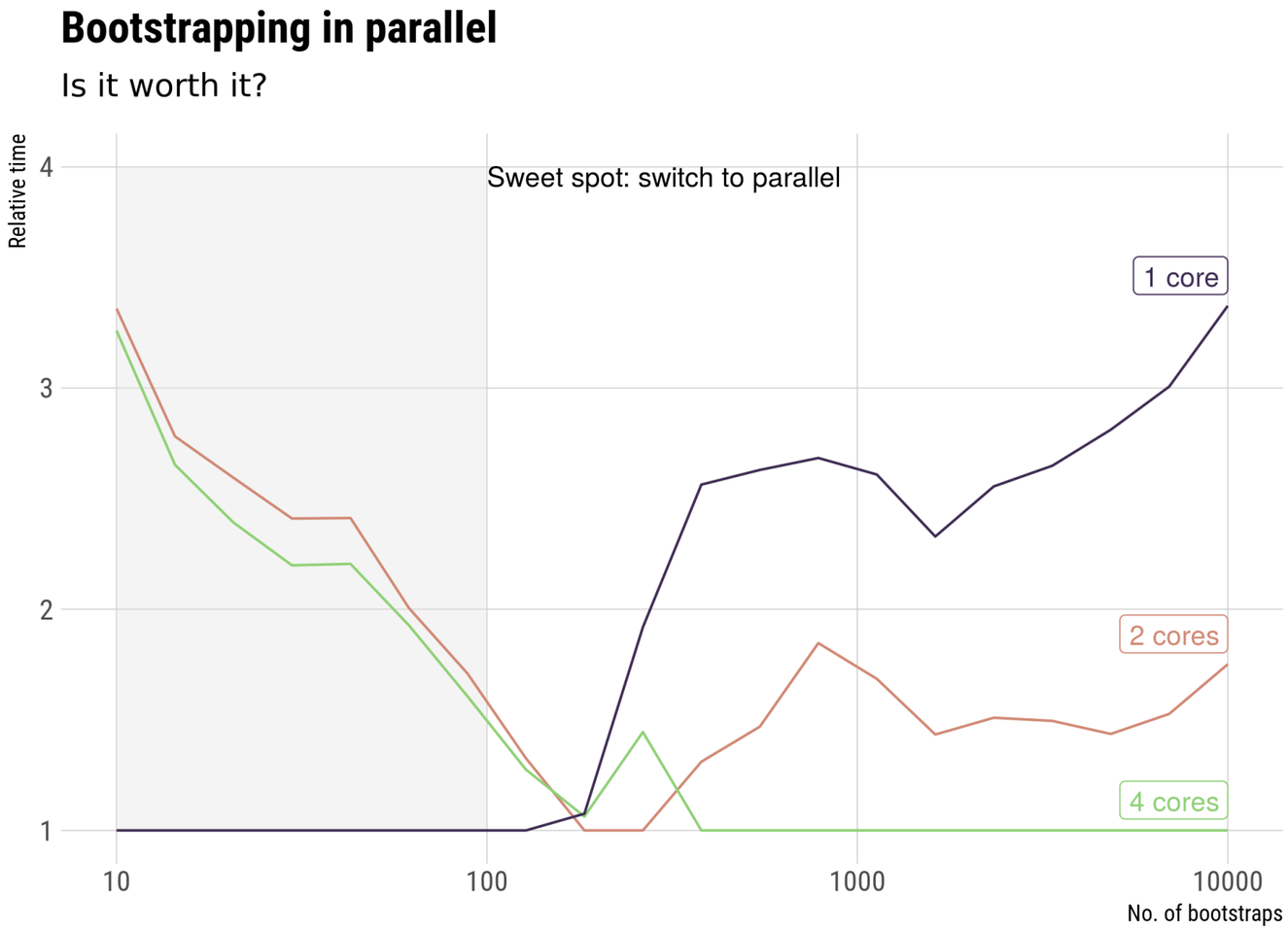
```
cl <- makeCluster(no_of_cores)
```

```
clusterExport(cl,
  c("bootstrap", "pokemon"))
```

```
parSapply(cl, 1:100,
  function(i) bootstrap(pokemon))
```

```
stopCluster(cl)
```

Timings


Bootstrapping in parallel
Is it worth it?
Relative time
4
3
2
1
Sweet spot: switch to parallel
1 core
2 cores
4 cores
10
100
1000
10000
No. of bootstraps