Debugging, Profiling and Optimization

RaukR 2022 • Advanced R for Bioinformatics

Ashfaq Ali (Slides Credit Marcin Kierczak)

NBIS, SciLifeLab

Contents

Topics of This Presentation

Code:

- **Debugging** -- my code does not run.
- **Profiling** -- now it does run but... out of memory!
- **Optimization** -- making things better.

Types of bugs

There are different types of bugs we can introduce:

- Syntax -- `prin(var1), mean(sum(seq((x + 2) * (y - 9 * b)))`
- Arithmetic -- `x/0` (not in R, though!) `Inf/Inf`
- Type -- `mean('a')`
- Logic -- everything works and produces seemingly valid output that is WRONG!

To avoid bugs:

- Encapsulate your code in smaller units (functions), you can test.
- Use classes and type checking.
- Test at the boundaries, e.g. loops at min and max value.
- Feed your functions with test data that should result with a known output.
- Use *antibugging*: `stopifnot(y <= 75)`

Arithmetic bugs


```
(vec <- seq(0.1, 0.9, by=0.1))
vec == 0.7
vec == 0.5
(0.5 + 0.1) - 0.6
(0.7 + 0.1) - 0.8 # round((0.7 + 0.1) , digits = 2) - 0.8
```

```
## [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [1] 0
## [1] -1.110223e-16
```

Beware of floating point arithmetics!

```
head(unlist(.Machine))
head(unlist(.Platform))
```

```
##     double.eps double.neg.eps    double.xmin
##   2.220446e-16   1.110223e-16  2.225074e-308
##    double.xmax    double.base  double.digits
##  1.797693e+308   2.000000e+00   5.300000e+01
##                     OS.type                    file.sep
##                      "unix"                         "/"
##                   dynlib.ext                         GUI
##                        ".so"                    "RStudio"
##                      endian                     pkgType
##                      "little" "mac.binary.big-sur-arm64"
```

Handling Errors


```
input <- c(1, 10, -7, -2/5, 0, 'char', 100, pi)
for (val in input) {
  (paste0('Log of ', val, 'is ', log10(val)))
}
```

```
## Error in log10(val): non-numeric argument to mathematical function
```

One option is to use the `try` block:

```
for (val in input) {
  val <- as.numeric(val)
  try(print(paste0('Log of ', val, ' is ', log10(val))))
}
```

```
## [1] "Log of 1 is 0"
## [1] "Log of 10 is 1"
## [1] "Log of -7 is NaN"
## [1] "Log of -0.4 is NaN"
## [1] "Log of 0 is -Inf"
## [1] "Log of NA is NA"
## [1] "Log of 100 is 2"
## [1] "Log of 3.14159265358979 is 0.497149872694133"
```

Handling Errors with `tryCatch`


```
for (val in input) {
  val <- as.numeric(val)
  result <- tryCatch(log10(val),
              warning = function(w) { print('Negative argument supplied. Negating.'); log10(-val) },
              error = function(e) { print('Not a number!'); NaN })
  print(paste0('Log of ', val, ' is ', result))
}
```

```
## [1] "Log of 1 is 0"
## [1] "Log of 10 is 1"
## [1] "Negative argument supplied. Negating."
## [1] "Log of -7 is 0.845098040014257"
## [1] "Negative argument supplied. Negating."
## [1] "Log of -0.4 is -0.397940008672038"
## [1] "Log of 0 is -Inf"
## [1] "Log of NA is NA"
## [1] "Log of 100 is 2"
## [1] "Log of 3.14159265358979 is 0.497149872694133"
```

Debugging -- Errors and Warnings

- An error in your code will result in a call to the `stop()` function that:
  - breaks the execution of the program (loop, if-statemetnt, etc.),
  - performs the action defined by the global parameter `error`.
- A warning just prints out the warning message (or reports it in another way).
- Global parameter `error` defines what R should do when an error occurs.

```
options(error = )
```

- You can use `simpleError()` and `simpleWarning()` to generate errors and warnings in your code:

```
f <- function(x) {
  if (x < 0) {
    x <- abs(x)
    w <- simpleWarning("Value less than 0. Taking abs(x)")
    w
  }
}
```

Debugging -- What are my Options?

- Old-school debugging: a lot of `print` statements
  - print values of your variables at some checkpoints,
  - sometimes fine but often laborious,
  - need to remove/comment out manually after debugging.
- Dumping frames
  - on error, R state will be saved to a file,
  - file can be read into debugger,
  - values of all variables can be checked,
  - can debug on another machine, e.g. send dump to your colleague!
- Traceback
  - a list of the recent function calls with values of their params,
- Step-by-step debugging
  - execute code line by line within the debugger

Option 1: Dumping Frames


```
options(error = quote(dump.frames("testdump", TRUE)))

f <- function(x) {
    sin(x)
}
f('test')
```

```
## Error in sin(x): non-numeric argument to mathematical function
```

```
options(error = NULL)
load("testdump.rda")
# debugger(testdump)
```

```
Message: Error in sin(x) : non-numeric argument to mathematical function
Available environments had calls:
1: f("test")

Enter an environment number, or 0 to exit
Selection: 1
Browsing in the environment with call:
f("test")
Called from: debugger.look(ind)
Browse[1]> x
[1] "test"
Browse[1]>
[1] "test"
Browse[1]>
```

Last empty line brings you back to the environments menu.

Option 2: Traceback


```
f <- function(x) {
  log10(x)
}

g <- function(x) {
  f(x)
}
g('test')
```

```
## Error in log10(x): non-numeric argument to mathematical function
```

```
> traceback()
2: f(x) at #2
1: g("test")
```

`traceback()` shows what were the function calls and what parameters were passed to them when the error occured.

Option 3: Debug step-by-step

Let us define a new function `h(x, y)`:

```
h <- function(x, y) {
  f(x)
  f(y)
  }
```

Now, we can use `debug()` to debug the function in a step-by-step manner:

```
debug(h)
h('text', 7)
undebug(h)
```

Option 3: Debug step-by-step cted.

`n` -- execute next line, `c` -- execute whole function, `q` -- quit debugger mode.

```
> debug(h)
> h('text', 7)
debugging in: h("text", 7)
debug at #1: {
f(x)
f(y)
}
Browse[2]> x
[1] "text"
Browse[2]> y
[1] 7
Browse[2]> n
debug at #2: f(x)
Browse[2]> x
[1] "text"
Browse[2]> n
Error in log10(x) : non-numeric argument to mathematical function
```

Profiling -- `proc.time()`

Profiling is the process of identifying memory and time bottlenecks in your code.

```
proc.time()
```

```
##     user   system  elapsed
##  379.248   53.761 1980.224
```

- `user time` -- CPU time charged for the execution of user instructions of the calling process,
- `system time` -- CPU time charged for execution by the system on behalf of the calling process,
- `elapsed time` -- total CPU time elapsed for the currently running R process.

```
pt1 <- proc.time()
tmp <- runif(n =  10e5)
pt2 <- proc.time()
pt2 - pt1
```

```
##    user  system elapsed
##   0.006   0.000   0.006
```

Profiling -- `system.time()`


```
system.time(runif(n = 10e6))
system.time(rnorm(n = 10e6))
```

```
##    user  system elapsed
##   0.230   0.008   0.239
##    user  system elapsed
##   0.254   0.006   0.261
```

An alternative approach is to use `tic` and `toc` statements from the `tictoc` package.

```
library(tictoc)
tic()
tmp1 <-runif(n = 10e6)
toc()
```

```
## 0.056 sec elapsed
```

Profiling in Action

Let's see profiling in action! We will define four functions that fill a large vector in two different ways:

```
fun_fill_loop1 <- function(n = 10e6, f) {
  result <- NULL
  for (i in 1:n) {
    result <- c(result, eval(call(f, 1)))
  }
  return(result)
}
```

```
fun_fill_loop2 <- function(n = 10e6, f) {
  result <- vector(length = n)
  for (i in 1:n) {
    result[i] <- eval(call(f, 1))
  }
  return(result)
}
```

Profiling in Action cted.

It is maybe better to use...vectorization!

```
fun_fill_vec1 <- function(n = 10e6, f) {
  result <- NULL
  result <- eval(call(f, n))
  return(result)
}
```

```
fun_fill_vec2 <- function(n = 10e6, f) {
  result <- vector(length = n)
  result <- eval(call(f, n))
  return(result)
}
```

Profiling our functions


```
system.time(fun_fill_loop1(n = 10e4, "runif")) # Loop 1
system.time(fun_fill_loop2(n = 10e4, "runif")) # Loop 2
system.time(fun_fill_vec1(n = 10e4, "runif"))  # Vectorized 1
system.time(fun_fill_vec2(n = 10e4, "runif"))  # Vectorized 2
```

```
##    user  system elapsed
##   7.750   3.240  11.067
##    user  system elapsed
##   0.168   0.015   0.184
##    user  system elapsed
##   0.001   0.001   0.000
##    user  system elapsed
##   0.001   0.000   0.001
```

The `system.time()` function is not the most accurate though. During the lab, we will experiment with package `microbenchmark`.

More advanced profiling

We can also do a bit more advanced profiling, including the memory profiling, using, e.g. `Rprof()` function.

And let us summarise:

```
summary <- summaryRprof('profiler_test.out', memory='both')
datatable(summary$by.self, options=list(pageLength = 10, searching = F, info = F))
#knitr::kable(summary$by.self)
```

Show 10 entries

	self.time	self.pct	total.time	total.pct	mem.total
"print.default"	36.47	97.2	36.47	97.2	36490.3
"runif"	0.67	1.79	0.67	1.79	1305.7
"eval"	0.21	0.56	37.46	99.84	38303.2
"fun_fill_loop2"	0.06	0.16	0.98	2.61	1783.6
"paste0"	0.05	0.13	0.05	0.13	8.6
"parent.frame"	0.02	0.05	0.02	0.05	41.9
"print"	0.01	0.03	36.48	97.23	36519.6
"w$get_new"	0.01	0.03	0.06	0.16	17.1
"is.list"	0.01	0.03	0.01	0.03	31.4
"is.pairlist"	0.01	0.03	0.01	0.03	25.6

Previous 1 Next

Profiling -- `profr` package

There are also packages available that enable even more advanced profiling:

```
library(profr)
Rprof("profiler_test2.out", interval = 0.01)
tmp <- table(sort(rnorm(1e5)))
Rprof(NULL)
profile_df <- parse_rprof('profiler_test2.out')
```

This returns a table that can be visualised:

Show 5 entries

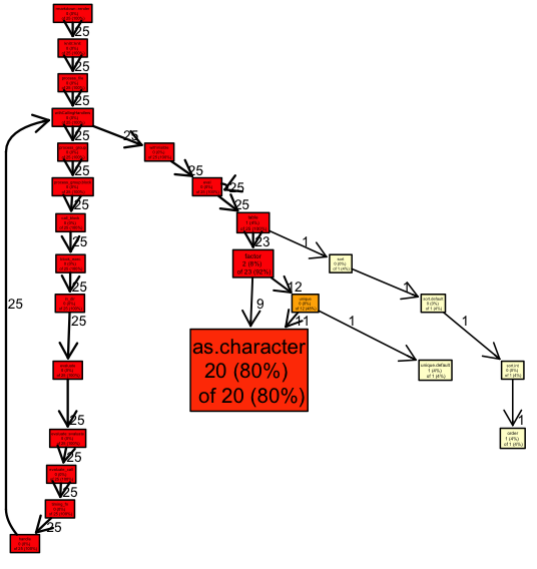
	level	g_id	t_id	f	start	end	n	leaf	time	source
1	1	1	1	table	0	0.62	1	false	0.62	base
2	2	1	1	sort	0	0.04	1	false	0.04	base
3	2	2	1	factor	0.04	0.6	1	false	0.56	base
4	2	3	1	as.integer	0.6	0.62	1	true	0.02	base
5	3	1	1	rnorm	0	0.02	1	true	0.02	stats

Previous 1 2 3 Next

Profiling -- `profr` package cted.

We can also plot the results using -- `proftools` package-

```
library("proftools") # depends on "graph" and "Rgraphviz" packages
profile_df2 <- readProfileData("profiler_test2.out")
plotProfileCallGraph(profile_df2, style = google.style, score = "total")
```



Profiling with `profvis`

Yet another nice way to profile your code is by using Hadley Wickham's `profvis` package:

```
library(profvis)
profvis({fun_fill_loop2(1e4, 'runif')
  fun_fill_vec2(1e4, 'runif')
  })
```

Profiling with `profvis` cted.

Flame Graph | Data | Options

<expr> | Memory | Time

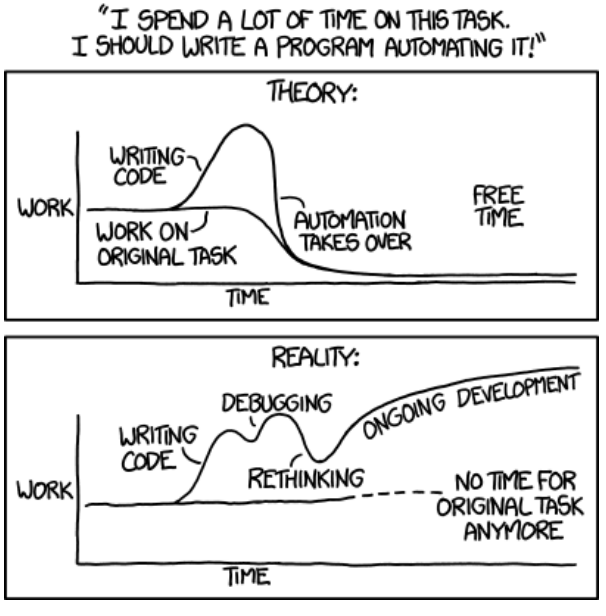
```
1  library(profvis)
2  profvis({fun_fill_loop2(1e4, 'runif')
3    c()
4    })
```

14.5 | 30

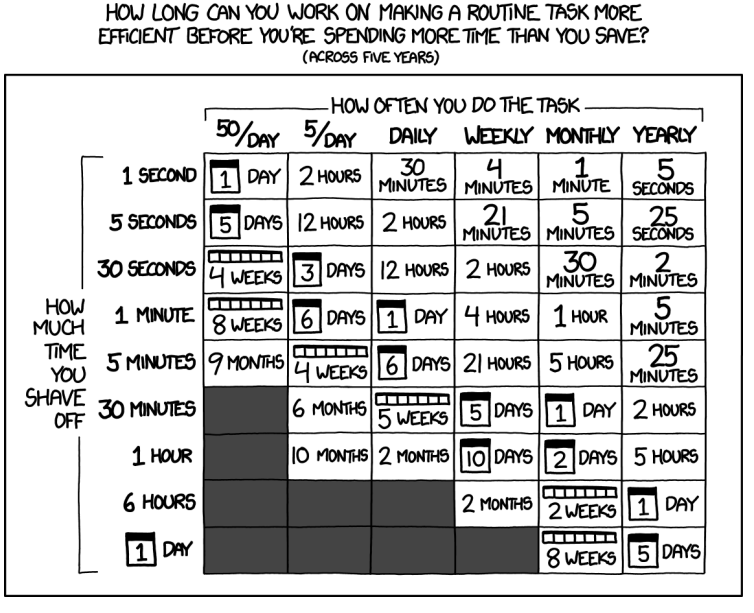
Optimizing your code

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 delulled 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



source: http://www.xkcd/com/1319



source: http://www.xkcd/com/1205

Ways to optimize the code

- write it in a more efficient way, e.g. use vectorization or `*apply` family instead of loops etc.,
- allocating memory to avoid copy-on-modify,
- use package `BLAS` for linear algebra,
- use `bigmemory` package,
- GPU computations,
- multicore support, e.g. `multicore`, `snow`
- use `data.table` or `tibble` instead of `data.frame`

Copy-on-modify


```
library(pryr)
order <- 1024
matrix_A <- matrix(rnorm(order^2), nrow = order)
matrix_B <- matrix_A
```

Check where the objects are in the memory:

```
address(matrix_A)
address(matrix_B)
```

```
## [1] "0x2873f8000"
## [1] "0x2873f8000"
```

What happens if we modify a value in one of the matrices?

```
matrix_B[1,1] <- 1
address(matrix_A)
address(matrix_B)
```

```
## [1] "0x2873f8000"
## [1] "0x287bfc000"
```

Avoid copying by allocating memory

No memory allocation

```
f1 <- function(to = 3, silent=F) {
  tmp <- c()
  for (i in 1:to) {
    a1 <- address(tmp)
    tmp <- c(tmp, i)
    a2 <- address(tmp)
    if(!silent) { print(paste0(a1, " --> ", a2)) }
  }
}
f1()
```

```
## [1] "0x11d01e8e0 --> 0x12849e648"
## [1] "0x12849e648 --> 0x12849e9c8"
## [1] "0x12849e9c8 --> 0x17253ee08"
```

Avoid copying by allocating memory cted.

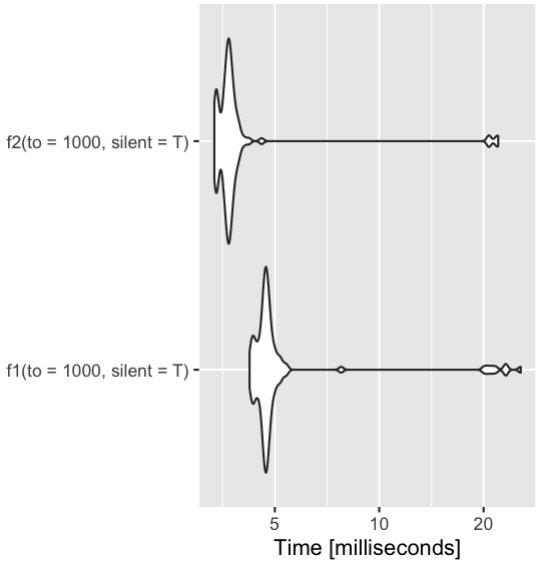
With allocation

```
f2 <- function(to = 3, silent = FALSE) {
  tmp <- vector(length = to, mode='numeric')
  for (i in 1:to) {
    a1 <- address(tmp)
    tmp[i] <- i
    a2 <- address(tmp)
    if(!silent) { print(paste0(a1, " --> ", a2)) }
  }
}
f2()
```

```
## [1] "0x173870898 --> 0x173870898"
## [1] "0x173870898 --> 0x173870898"
## [1] "0x173870898 --> 0x173870898"
```

Allocating memory -- benchmark.

```
library(microbenchmark)
benchmrk <- microbenchmark(f1(to = 1e3, silent = T),
                           f2(to = 1e3, silent = T),
                           times = 100L)
autoplot(benchmrk)
```



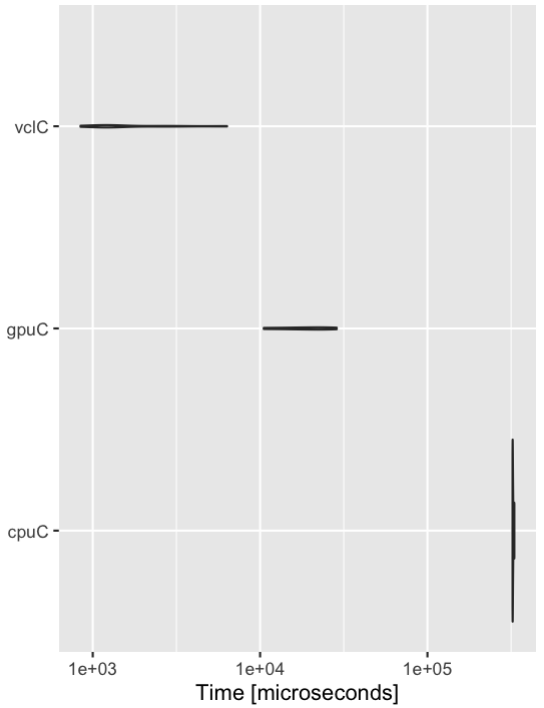
GPU

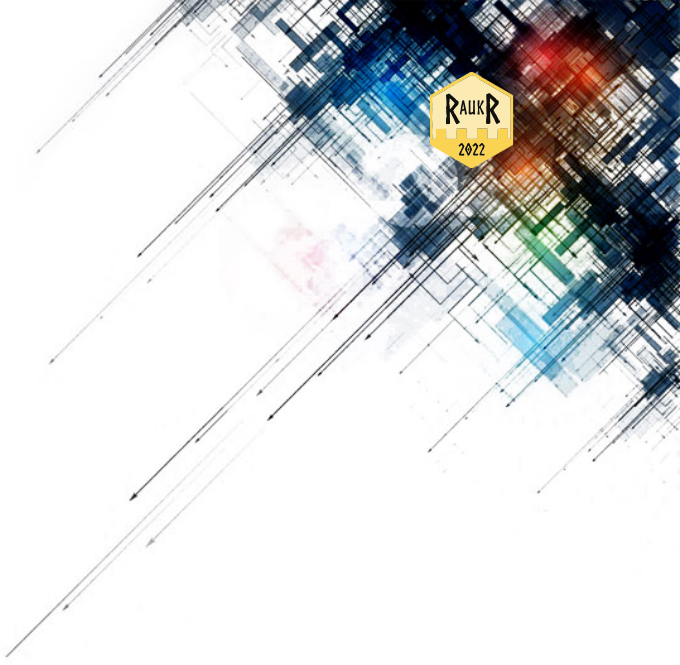

```
library(gpuR)
library(microbenchmark)
A = matrix(rnorm(1000^2), nrow=1000) # stored: RAM, computed: CPU
B = matrix(rnorm(1000^2), nrow=1000)
gpuA = gpuMatrix(A, type = "float") # stored: RAM, computed: GPU
gpuB = gpuMatrix(B, type = "float")
vclA = vclMatrix(A, type = "float") # stored: GPU, computed: GPU
vclB = vclMatrix(B, type = "float")
bch <- microbenchmark(
  cpuC = A %*% B,
  gpuC = gpuA %*% gpuB,
  vclC = vclA %*% vclB,
  times = 10L)
```

More on Charles Determan's Blog.

GPU cted.

```
autoplot(bch)
```





Thank you. Questions?

R version 4.1.2 (2021-11-01)
Platform: aarch64-apple-darwin20 (64-bit)
OS: macOS Monterey 12.1

Built on : 09-Jun-2022 at 20:08:55
2022 • SciLifeLab • NBIS • RaukR