Lecture 12 – Profiling and Optimizing code in R

Learning Objectives:

3. Learn the basic principles of software design.

   3.4. Learn about optimization and profiling code.

What is ‘Optimization’?

- any modifications that you make to code that improves quality, efficiency, and/or speed
- Can have a variety of goals: execution speed, memory usage, minimizing inputs, making the program smaller
- We’ll cover two: speed (today) and memory (Monday)
- Often these work hand-in-hand!
- Tradeoffs exist between speed and efficiency or flexibility. Thinking carefully about what ***you want out of optimization*** will reduce the amount of time you spend optimizing code.

Should you optimize?

- Should you make it faster? Is it worth the time to optimize it?
- How much faster does it need to be?
- How much time should you devote to making it faster?

Why is R so slow?

- *Extreme dynamism* (very dynamic and flexible, gives up speed)
- *Name lookup*: lexical scoping during function calling (like, everything is a function)
- *Lazy evaluation*: promise objects creates extra stuff that slows everything down

… but mostly because code is not efficient.

The Optimization Process:

1. Find the biggest time suck.

2. Try to fix that one (might not work).

3. Start over until you're satisfied.

How do you find this?

Profile!

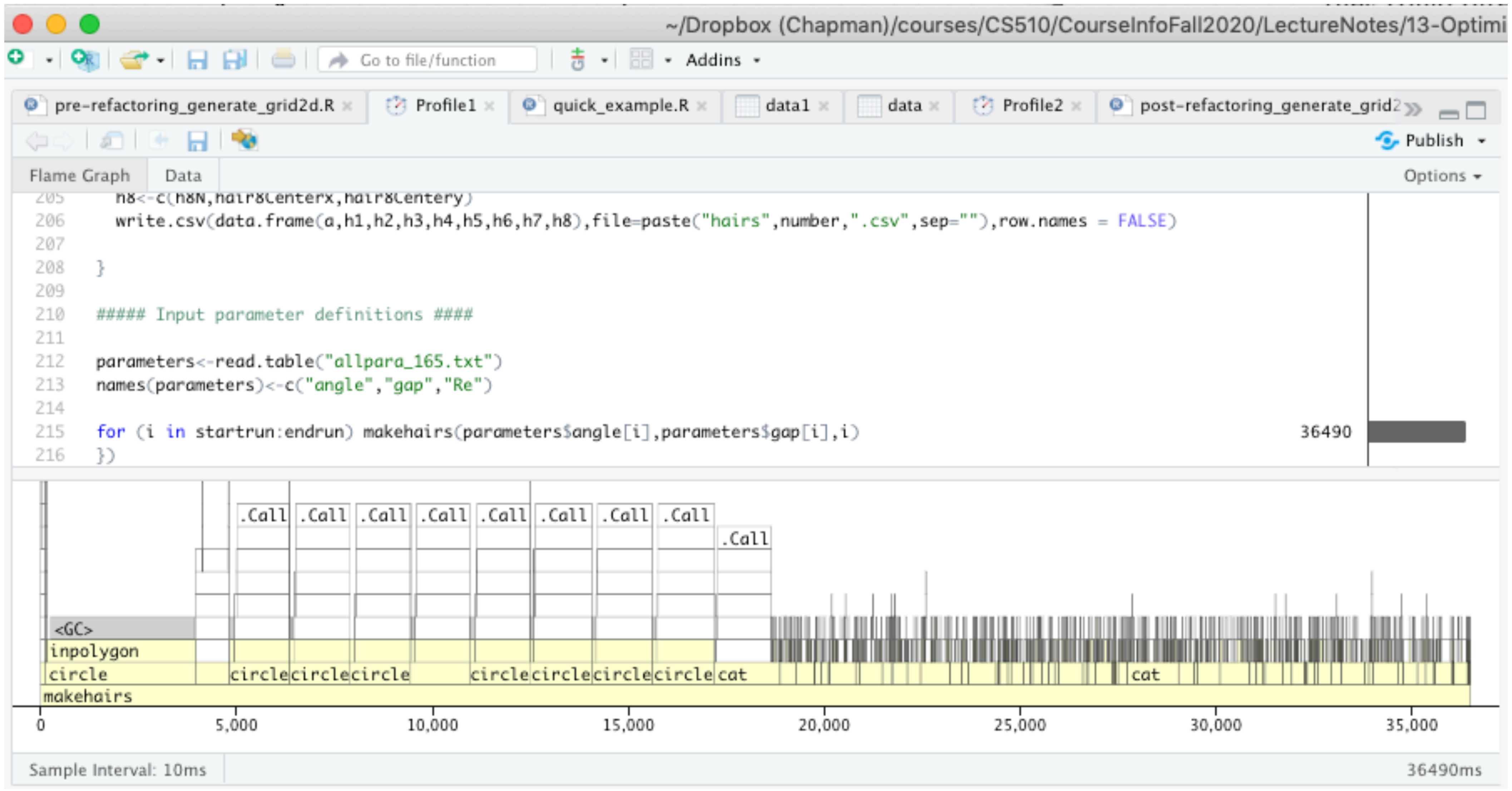
Try not to rely on your instincts.

Just Profile It

Profilers in R

`profvis` package: RStudio's built-in profiler.

ProfilingExample in 12-Optimization



```
  h8<-c(h8N,hair8centerx,hair8centery)
  write.csv(data.frame(a,h1,h2,h3,h4,h5,h6,h7,h8),file=paste("hairs",number,".csv",sep=""),row.names = FALSE)

}

##### Input parameter definitions ####

parameters<-read.table("allpara_165.txt")
names(parameters)<-c("angle","gap","Re")

for (i in startrun:endrun) makehairs(parameters$angle[i],parameters$gap[i],i)        36490
})
```

`microbenchmark` package: high-precision profiler for R.

```
> x = runif(100)
> microbenchmark(sqrt(x), x^0.5)
```

Common and Quick Strategies for Improving Speed

General advice:

- Keep a record
- Generate a representative test case
- Set a target speed / time amount

Specific strategies:

- Look for existing solutions
- Do less work with more appropriate functions
- Vectorize!
- Parallelize (in a few weeks), avoid copies (next time)
- Byte-code compile
- Rewrite in something faster

Look for existing solutions

- Don't reinvent the wheel, do yourself a google.

  - Stack Overflow
  - https://www.rseek.org Google for R!
  - CRAN task views cran.rstudio.com/web/views: list of packages in CRAN that are associated with specific tasks.
  - Reverse dependencies of Rcpp: cran.r-project.org/web/packages/Rcpp (might find something already written in C++)

Remember: if you publish based on code that relies on someone else's package, be sure to cite them!!!

Do as little as possible: look for tailored functions or tailor your functions!

- `rowSums()`, `colSums()`, `rowMeans()`, and `colMeans()` are faster than equivalent functions on vectors. These vectorize with `apply()`, so they are faster.

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

- Use `any(x == 10)` instead of `10 %in% x`.

- `read.csv()` goes faster if you specify `colClasses`.

- `factor()` is faster if you specify known levels

- `unlist()` is faster if you the flag `use.names=FALSE`

Vectorize!

- Avoid loops in favor of vectorized calculations.
  - Vectorized calculations for many functions in base push loops into C (which is much faster).
  - Makes many problems easier anyway
  - Vectorized subsetting can be lots faster. Using this in a single step is faster than breaking it up (such as `x[is.na(x)]<-0`) because of the looping issue.
  - Be aware of vectorize functions, `cumsum()`, `diff()`, and the above
  - But don't "torture" your function into vectorize version, sometimes it just won't work.

Byte-code Compilation

- Byte code compilation can increase speed of some code.

  - Compile functions using `cmpfun()` in compiler package

  - Sometimes will help right away, sometimes won't, but it really doesn't take much effort to do (so not much time lost).

  - Can get you 10-15% improvement in speed.

  - (All base functions are byte-compiled by default.)

Other stuff

- Improve memory usage (we’ll talk about this in a bit)
- Parallelize (we’ll also talk about this)
- Read about what other people have done!
- Read books on programming in R! Patrick Burns’ *R Inferno* has a lot of common traps.

Try it out!

- Example in Advanced R: 17.9 (page 371), improving speed of the t-test.

Imagine we have 1000 experiments (rows), each of which collects data on 50 individuals (columns). The first 25 individuals in each experiment are assigned to group 1, the rest to group 2. We'll generate some random data and then run the t test using base t.test() and apply() to store the results. **See opt_t-test.R for the start of this code.**

Optimize the t.test() function! The book example steps you through this, but try to do each step on your own.

More Information

https://github.com/jennybc/code-smells-and-feels – “Code Smells and Feels” by Jenny Bryan, a talk at the UseR conference 2018

http://silab.fon.bg.ac.rs/wp-content/uploads/2016/10/Refactoring-Improving-the-Design-of-Existing-Code-Addison-Wesley-Professional-1999.pdf – Refactoring: Improve the Design of Existing Code by Martin Fowler