Statistics 360: Advanced R for Data Science

Lecture 6

Brad McNeney

Debugging (Ch22) and Measuring performance (Ch 23)

- Reading: Text, Chapters 22 and 23
- Topics on debugging:
  - overview of debugging
  - tracing execution with `traceback()`
  - interactive debugging with `debug()` and `browser()`
  - non-interactive debugging: `dump.frames()` and printing
  - test cases to detect future bugs
- Topics on measuring performance:
  - profiling
  - microbenchmarking
  - final thoughts

Debugging

Overview

- Focus on the easy part of debugging: finding and fixing the source of unexpected errors.
  - Mistakes that give incorrect results but throw no errors are harder to find.
- Workflow tips for finding and fixing errors
  - Google it: If you don't understand the error message, try pasting it into a google search.
  - Make a small self-contained example (reproducible example, a.k.a. reprex).
  - Find it with tools like `traceback()`, `debug()` and `browser()`.
  - Fix it and make a test case to alert you if you accidentally re-introduce the bug.

Reproducible examples

- Reproducible means including any source code, data and library calls so that the code can run as it did when the error was triggered.
- Next reduce the code to a minimal example that triggers the problem.
  - For example, remove lines of code, compute on a smaller R object, use build-in data.
- The act of creating the reprex may show you the error.
- If not, you are in a position to ask for help from a class-mate, mailing list or stack overflow.
- I find it hard to construct reprexs without first finding the lines that throw the error ...

Tracing execution

- After an error, you can use `traceback()` to see the sequence of function calls ("call stack") that lead to the error.
  - The numbers in each entry are supposed to be line numbers of the call in the calling function, but they usually just confuse me ...

```
f <- function(x) { g(h(x)) }
g <- function(x) {
  x
}
h <- function(x) {
  if(!is.numeric(x)) stop("x must be numeric")
}
# f("cat") # uncomment to run
# traceback()
```

Interactive debugging

- Main tools are `browser()` and `debug()`.
- Stop and step through function execution.
  - Can print variables and execute R commands to investigate

```
h <- function(x) {
  browser()
  if(!is.numeric(x)) stop("x must be numeric")
}
#f("cat")
```

browser commands

- `n` executes the next step. Use `print(n)` to print a variable named `n`.
- `s` is like `n` but will step into a function call.
- `f` finishes execution of the current loop or function.
- `c` leaves interactive debugging and continues regular execution.
- Enter (Return) repeats the last browser command
- `Q` completely exits the function.

debug()

```r
h <- function(x) {
  if(!is.numeric(x)) stop("x must be numeric")
}
# debug(f)
# f("cat")
# undebug(f)
# debug(g)
# f("cat")
# undebug(g)
# debug(h)
# f("cat")
# undebug(h)
```

Non-interactive debugging

- You can insert `print()` or `cat()` statements to see values of variables in your code if you find the trace too confusing and browser too time-consuming.

Test cases

- After you find and fix a bug it is a good idea to devise a test of your code that will flag the problem if you ever accidentally re-introduce it.
- If you are writing an R package you should investigate the `testthat` package, which helps you compile and run "unit" tests on small pieces of your code.

```r
f <- function(x) { x + 3 }
# test
f(3) # should return 6
```

```
## [1] 6
```

Measuring performance

Measuring performance

- When you write code you develop an intuition about what parts will run slowly – don't trust this!
- As statisticians we know that the only thing you can trust is data.
- Profiling and benchmarking are ways to collect data on your code

```
library(profvis) #visualize profiling data
library(bench) # benchmarking tools
```

Profiling

- R uses a statistical profiler that records the call stack at small intervals.
  - Read the call stack from right to left

```
f <- function() {pause(0.1);g();h()} # pause() is from profvis
g <- function() {pause(0.1);h()}
h <- function() {pause(0.1)}
Rprof()
f()
```

```
## NULL
```

```
Rprof(NULL) # Now view Rprof.out
```

```
sample.interval=20000
"pause" "f"
"pause" "f"
"pause" "f"
"pause" "f"
"pause" "f"
"pause" "g" "f"
"pause" "g" "f"
"pause" "g" "f"
"pause" "g" "f"
"pause" "g" "f"
"pause" "h" "g" "f"
"pause" "h" "g" "f"
"pause" "h" "g" "f"
"pause" "h" "g" "f"
"pause" "h" "g" "f"
"pause" "h" "f"
"pause" "h" "f"
"pause" "h" "f"
"pause" "h" "f"
"pause" "h" "f"
```

Summary of profile

```
# summaryRprof() # uncomment and run
```

Visualize profile

- `profvis()` gives a nicer summary of the profiling.
- Two panels:
  - top shows the source code with graphs depicting memory use and execution time
  - bottom is a “flame graph” showing the call stack, read bottom to top

```
source("lec6profiling.R") # profiler will refer to this source f
# profvis({ f() }) # Or choose Profile from RStudio
```

Memory profiling

- The `profvis()` output also includes information about memory usage.
- Illustrate with an example from the week 2 exercises where we built a dataset of 500×1000 observations.
  - included in source file `lec6profiling.R`

```
# profvis({ bigd1() })
# profvis({ bigd2() })
# profvis({ bigd3() })
```

Notes on memory profiling

- The grey bars in the bar and flame graphs show memory being freed by the garbage collector (notice the `<GC>` when you hover over the grey bars in the flamegraph).
- Memory claiming and freeing in bigd1 $>$ bigd2 $>$ bigd3.

Microbenchmarking

- We have done benchmarking on bigd1/2/3 with `system.time()`.
- For small chunks of code that take less time, `system.time()` is less useful.
- Microbenchmarking measures performance of code chunks that run in very small time increments.
- To do this, the `bench` package uses a high precision timer.

bench

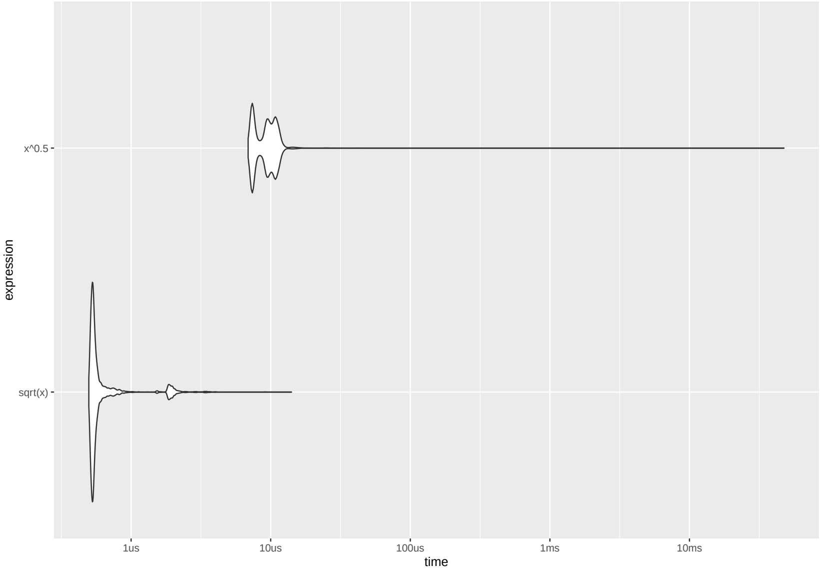
- Times the expression in multiple runs.
  - At least one run and at most the number of runs it can do in 1/2 sec.
- Output displayed as a summary table or graph.

```
library(bench)
x <- runif(100)
lb <- mark(
  sqrt(x),
  x ^ 0.5
)
lb
```

```
## # A tibble: 2 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 sqrt(x)    497.91ns    539ns  1386054.      848B        0
## 2 x^0.5        6.89us   9.33us   107423.      848B     10.7
```

```r
plot(lb,type="violin")
```

```
## Loading required namespace: tidyr
```



Interpretation

- Notice the units in the output table, microseconds for `sqrt` and nanoseconds for `x^{0.5}`
- Run-time summary statistics include min, mean, median, max, and itr/sec.
- `mem_alloc` and `gc/sec` related to memory usage
- There are other columns in the output not shown – see `?mark` for details.
- The bumps in the violin plot may indicate that your computer was doing something else during some of the runs
- Notice the highly skewed distribution – median is more useful than mean

Final thoughts

- Avoid the temptation to let performance considerations dominate your code development and lead you to profile and benchmark extensively.
- Remember that the most important performance improvement is code that gives the right answer.