Performance & Bugs

Cecina Babich Morrow

2023-10-31

Debugging

Debugging refers to the process of fixing "bugs", i.e. errors in code causing erroneous results or errors. There are many potential strategies and tools for carrying out the debugging process.

From a philosophical standpoint, we can think of debugging using the process of the scientific method. In this framework, you can run experiments on the code using toy examples to check whether the output matches the expected hypothesis. If the bug in question produces an error message, you can start with this message to try to understand the cause of the error. Best case scenario, the error message is informative and can point you in the direction of what needs to be fixed. If the error message is more cryptic, debugging requires further investigation. To run experiments to determine the nature of the bug, you should try to create a reproducible, minimal working example: the simplest possible example that produces the error and can be reproduced (both by you and by others, if necessary). This helps avoid a situation where you are trying to test your code on a very complex set of data, for example, where the issue could arise from any number of places. Additionally, by creating a reproducible example, you can share your issue with other people, e.g. on StackOverflow or by creating a GitHub issue if you identify a bug in someone else's code.

`traceback`

The `traceback` function can aid in debugging by showing the series of functions called, including the one that resulted in an error. This is useful in situations where the error message received is not informative. For a simple example:

```
# Simple addition function
add_numbers <- function(x, y) {
  result <- x + y
  return(result)
}

# Function that calls add_numbers
outer_function <- function(x) {
  result <- add_numbers(x, 20)
  return(result)
}

# Function that intentionally produces an error
outer_function("a")
```

```
## Error in x + y: non-numeric argument to binary operator
```

```
traceback()
```

```
## No traceback available
```

```
2: add_numbers(x, 20) at #2
1: outer_function("a")
```

This toy example shows how `traceback()` displays the chain of functions used that resulted in the error.

`browser`

The `browser()` function is also a useful tool for debugging that opens an interactive debugging environment. You can insert `browser()` into your code where you want to pause execution and examine the code more fully. This will open up an interactive environment that allows you to run R code to examine the current value of variables, explore the results of the functions, etc. There are also some special commands:

- Next `n`: evaluates the next step of the function
- Step `s`: similar to `n`, but if the next line is a function, it will step into that function
- Finish `f`: finish the current loop or function
- Continue `c`: get out of the interactive debugging environment and execute the rest of the function as usual
- Stop `Q`: stop debugging and cease running the function
- `where`: print a stack trace of all active function calls

As an alternative, you can also use `debug(<function_name>)` on a given function to initiate `browser` for each step of that function. Note that you will need to run `undebug(<function_name>)` once you are done so that you can run that function again later without entering the debugger.

Debugging resources

The following resources have helpful information about the debugging process:

- Advanced R: Chapter 22 Debugging
- What They Forgot to Teach You About R: Chapter 13 Debugging R code

Profiling

Profiling allows us to see where the code is spending the most time. This helps prioritize where to spend effort optimizing: in order to improve overall performance, it is most impactful to improve the performance of the part of the code that takes the most computation time. Additionally, in situations where there is a trade-off between readability and performance, profiling can show whether the increase in overall performance would be worth any loss in readability. In R, we can use the `profvis` package to profile. This package is a statistical profile, which regularly interrupts the operating system and checks what code is being executed, determining where the code is spending most of the time.

In R, we want to avoid for loops that don't contain substantial amounts of computation within each iteration of the loop. In this case, it is almost always better to vectorize the code instead in order to eliminate unnecessary for loops. I can compare the results of profiling using 2 for loops compared to a vectorized approach:

```r
library(profvis)
source("../R/mvn_functions.R")
plot_mu <- matrix(c(2,1), ncol = 1)
plot_Sigma <- matrix(c(1,0.5,0.5,1), ncol = 2)
```

The following code is not evaluated because the interactive output cannot be displayed in a pdf, but this code does enable the comparison of profile for the two functions:

```r
# Profile the 2 for loop function
profvis({mvn_generator_2_for_loops(dimension = 2,
                                   mu = plot_mu,
                                   Sigma = plot_Sigma,
                                   n_samples = 100000)})
# Profile the 2 for loop function
profvis({mvn_generator_vector(dimension = 2,
                                   mu = plot_mu,
                                   Sigma = plot_Sigma,
                                   n_samples = 100000)})
```

Performance

R code typically has worse performance than code written in a language like C. However, R has several useful features to minimize the chance of making an error.

Memory management

Pass by value vs. pass by reference

R uses pass-by-value semantics, where the values of variables are passed to functions, meaning that a function receives a copy of the variable's value rather than a reference to the original variable. This approach can impact performance, particularly in terms of memory management. R does some performance optimization under the hood, such as ensuring that arrays are not duplicated when they are updated in a for loop. However, it is necessary to be cautious when writing code to get the benefits of these optimization and avoid unneccessary copies of objects being made.

Column-major storage

R uses column-major order, meaning that consecutive elements of a column are stored adjacent to one another in memory (rather than consecutive elements of a row). This means that filling a matrix in R column by column is faster than row by row, since it is faster to read / write memory in contiguous blocks.