Debugging and profiling in R

Feng Li
feng.li@cufe.edu.cn

School of Statistics and Mathematics
Central University of Finance and Economics

August 15, 2018

Revised on August 15, 2018

The basic concepts of debugging

- **Debugging** is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware, thus making it behave as expected.
- A **debugger** or debugging tool is a computer program that is used to test and debug other programs (the "target" program)
- Debugging involves numerous aspects including **interactive debugging**, **control flow**, **integration testing**, **log files**, **monitoring** (application, system), **memory dumps**, **profiling**, **Statistical Process Control**, and special design tactics to improve detection while simplifying changes.

Typical debugging process

- Normally the first step in debugging is to attempt to reproduce the problem.
- After the bug is reproduced, the input of the program may need to be simplified to make it easier to debug
- After the test case is sufficiently simplified, a programmer can use a debugger tool to examine program states (values of variables, plus the call stack) and track down the origin of the problem(s). Alternatively, tracing can be used

Debugging tools in R

- The simplest version:
  `cat(); print()`
- `browser()`
  At the browser prompt the user can enter commands or R expressions, followed by a newline. The commands are
  - 'c' (or just an empty line, by default) exit the browser and continue execution at the next statement.
  - 'n' enter the step-through debugger if the function is interpreted. This changes the meaning of 'c': see the documentation for 'debug'. For byte compiled functions 'n' is equivalent to 'c'.
  - 'where' print a stack trace of all active function calls.
  - 'Q' exit the browser and the current evaluation and return to the top-level prompt.
  - `> options(browserNLdisabled = TRUE)`
- `trace() traceback()`
- if control flow
- `ls()`
- `try() tryCatch()`

Why profiling?

- Find the computational bottom-neck of your code.
- Fine the memory bottom-neck of your code.

Profiling R code for speed

- Check computing time of a piece of code: `proc.time()`.
- Profiling works by recording at fixed intervals (by default every 20 msecs) which line in which R function is being used, and recording the results in a file.
- The R profiling procedure

```
Rprof("myprofile.out") # Open the profile log file
##
.... ## Some code you want to profile
##
Rprof(NULL) # Close the profile log

summaryRprof("myprofile.out") # summarize the results
```

Profiling R code for memory use I

- Measuring memory use in R code is useful either when the code takes more memory than is conveniently available or when memory allocation and copying of objects is responsible for slow code.
- Garbage collection: gc()

```
>gc()
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 311043 16.7     597831 32.0   597831 32.0
Vcells 761909  5.9    1445757 11.1  1137162  8.7
```

- **Vcells** used to store the contents of vectors
- **Ncells** used to store everything else, including all the administrative overhead for vectors such as type and length information. In fact the vector contents are divided into two pools.
- The sampling profiler Rprof described in the previous section can be given the option `memory.profiling=TRUE`.

Profiling R code for memory use II

```
Rprof("myprofile.out", memory.profiling=TRUE) # Open the profile
##
.... ## Some code you want to profile
##
Rprof(NULL) # Close the profile log

summaryRprof("myprofile.out") # summarize the results
```

- Memory profiling requires R to have been compiled with `--enable-memory-profiling`, which is not the default, but is currently used for the OS X and Windows binary distributions.

Suggested Reading

- Jones (2009), **Chapter 3.7, 5.6, 8.3, 9.3, 9.5**