Debug and profiling in R

ANF R

Vincent Miele

CNRS

08/10/2015

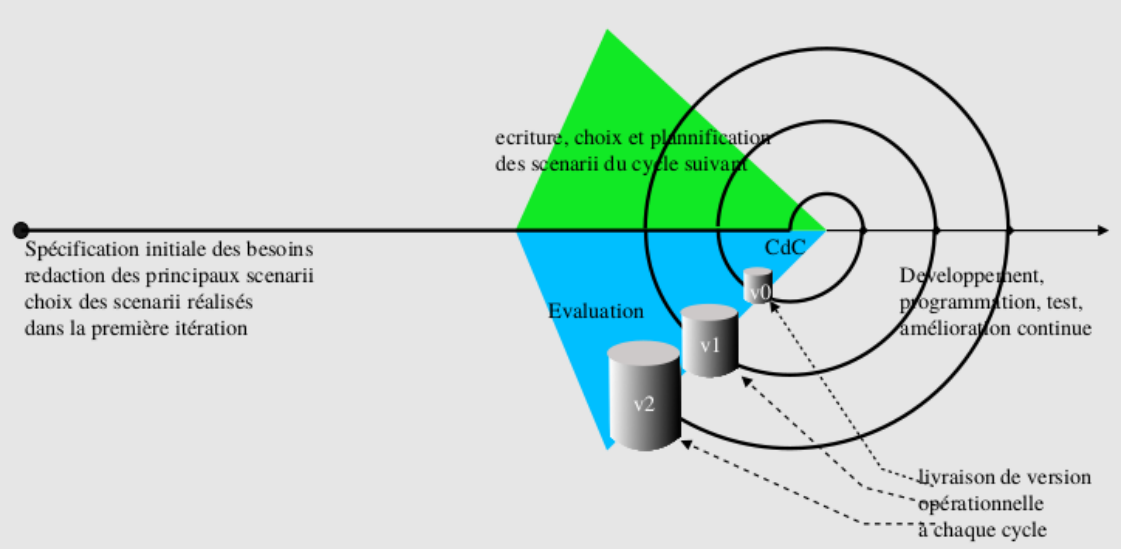
Roadmap

Simple rules
- Herb Sutter
- Agile manifesto & co

Debug
- Principle
- Anticipating bugs
- Tracking bugs with appropriate tools - pure R code
- Tracking bugs with appropriate tools - R/C++ code

Profiling
- Principle
- Recording time and space requirements
- Performance measurements with appropriate tools - pure R code
- Performance measurements with appropriate tools - R/C++ code

- Correctness (this course), simplicity and clarity come first
- Don't optimize prematurely (this course)
- Give one entity one cohesive responsability
- Use a version control system



- Repeat very short development cycles
- Write tests first (this course, quickly)
- Adopt naming conventions and use explicit names
- Don't hesitate to refactor your code (this course)
- Submit frequent deliveries

Roadmap

Simple rules
- Herb Sutter
- Agile manifesto & co

Debug
- Principle
- Anticipating bugs
- Tracking bugs with appropriate tools - pure R code
- Tracking bugs with appropriate tools - R/C++ code

Profiling
- Principle
- Recording time and space requirements
- Performance measurements with appropriate tools - pure R code
- Performance measurements with appropriate tools - R/C++ code

Finding a bug is a process of confirming the many things that you believe are true - until you find one which is not true. (Norm Matloff)

1. Realise you have a bug
2. Make it repeatable
3. Figure out where it is
4. Fix it and test

1. Tests
2. Tests
3. Debugger
4. Tests

Debug

Anticipating bugs

Print-like method : `stop()`, `message()`, `warning()`

Condition handling method : `try()`, `tryCatch()`

Defensive programming or "failing fast", i.e. never suppose anything

Debug

Tracking bugs with appropriate tools - pure R code

- Determining the sequence of calls that lead to an error `where`
- Consulting the content of variables `print`

Global Environment

Data	
distance.matrix	Large matrix (1000000 elements, 7.7 Mb)
points	num [1:1000, 1:2] 0.3748 0.9824 0.425 0.677 0.0518 ...
Values	
clusters	List of 4
N	1000
x	num [1:250] 0.3748 0.9824 0.425 0.677 0.0518 ...
y	num [1:250] 0.609 0.623 0.789 0.701 0.301 ...

- Setting breakpoints `break`
- Execute the code step-by-step `next`,`step`

TP Exercise(s) 1

Debug

Tracking bugs with appropriate tools - R/C++ code

Much more complicated...

- Bioconductor guidelines

DEMO gdb

DEMO valgrind

Roadmap

Simple rules
- Herb Sutter
- Agile manifesto & co

Debug
- Principle
- Anticipating bugs
- Tracking bugs with appropriate tools - pure R code
- Tracking bugs with appropriate tools - R/C++ code

Profiling
- Principle
- Recording time and space requirements
- Performance measurements with appropriate tools - pure R code
- Performance measurements with appropriate tools - R/C++ code

Programmers waste enormous amouts of time thinking about, or worrying about, the spedd of noncritical pats of their programs. (Donald Knuth)

Writing faster but incorrect code... Writing code that you think faster, but is actually no better. (Hadley Wickam)

1. Find the bottleneck
2. Refactor the code
3. Consider using third party librairies or packages
4. Loop to 1

1. Profiler
2. Advanced R or Rcpp
3. Web browser, support from R-help, calcul-list or colleagues, forums
4. Loop to 1

Print-like method : `sys.time()`, `microbenchmark()`, `mem_used()`, `top` (Unix only)

It is mandatory to have a methodology for measuring performance : small to large datasets or problem dimensions + estimation of the theoritical complexity

TP Exercise(s) 2,3

R proposes a *sampling* profiler called *Rprof* : it stops the execution of the code every few milliseconds and writes in a file which function is currently executing.

For `Rprof` in parallel, the workers (not the master) launch Rprof and specify different files.

Now, let's understand the *call graph* !

$by.self

	self.time	self.pct	total.time	total.pct
".C"	35.60	32.23	35.60	32.23
"data.frame"	10.36	9.38	59.32	53.70
"deparse"	6.96	6.30	23.74	21.49
"FUN"	6.38	5.78	110.46	100.00
".deparseOpts"	6.20	5.61	9.70	8.78
"as.data.frame.numeric"	4.94	4.47	32.08	29.04
"as.data.frame"	3.94	3.57	36.02	32.61
"match"	3.22	2.92	14.94	13.53
"pmatch"	3.16	2.86	3.50	3.17
"as.list"	2.92	2.64	3.86	3.49
"length"	2.54	2.30	2.54	2.30
"make.names"	2.50	2.26	2.86	2.59
"mode"	2.10	1.90	13.06	11.82
"paste"	2.06	1.86	25.80	23.36
"row.names"	1.80	1.63	2.54	2.30
"unlist"	1.58	1.43	3.12	2.82
"meanRuptR_c"	1.34	1.21	22.88	20.71
".row_names_info"	1.30	1.18	1.30	1.18
".set_row_names"	1.20	1.09	1.20	1.09
"names"	1.10	1.00	1.10	1.00
"%in%"	1.06	0.96	15.46	14.00

$by.total

	total.time	total.pct	self.time	self.pct
"FUN"	110.46	100.00	6.38	5.78
"lapply"	110.46	100.00	0.08	0.07
"multiseg"	110.46	100.00	0.00	0.00
"multisegmean"	110.46	100.00	0.00	0.00
"standardGeneric"	110.46	100.00	0.00	0.00
"sapply"	95.54	86.49	0.00	0.00
"multisegout"	92.78	83.99	0.02	0.02
"data.frame"	59.32	53.70	10.36	9.38
"as.data.frame"	36.02	32.61	3.94	3.57
".C"	35.60	32.23	35.60	32.23
"as.data.frame.numeric"	32.08	29.04	4.94	4.47
"force"	26.16	23.68	0.36	0.33
"paste"	25.80	23.36	2.06	1.86
"deparse"	23.74	21.49	6.96	6.30
"meanRuptR_c"	22.88	20.71	1.34	1.21
"%in%"	15.46	14.00	1.06	0.96
"unisegmean"	15.22	13.78	0.00	0.00
"segmeanCO"	15.10	13.67	0.00	0.00
"match"	14.94	13.53	3.22	2.92
"colibriR_c"	14.72	13.33	0.00	0.00
"mode"	13.06	11.82	2.10	1.90

TP Exercise(s) 4,5,6

Performance measurements with appropriate tools - R/C++ code

See forums or read "Writing R extensions" (that suggests the use of `oprofile`)

```
samples  %        symbol name
61898    19.0842  Liste::resetMaillonBorders(Polynome2*)
49520    15.2678  Polynome2::roots(double)
38476    11.8628  Polynome2::minOrMax(double*, double*, int*)
33504    10.3298  Liste::removeDoublon()
17497     5.3946  Polynome2::add(double, double, double)
16009     4.9358  Liste::getPolynome()
9972      3.0745  Liste::computeMinOrMax(double*, int*)
9279      2.8609  Liste::add(double, double, double)
9141      2.8183  Liste::getNext()
7334      2.2612  colibri_c(double*, int*, int*, double*, double*, int*, double*)
6953      2.1437  Liste::resetAllBorders(Polynome2*)
6241      1.9242  Polynome2::reset(double, double, double, int)
5754      1.7741  Liste::computeRoots(double)
5413      1.6689  Polynome2::delta(double)
5398      1.6643  Liste::checkForDoublon()
5386      1.6606  Polynome2::getRacine2()
4740      1.4614  Liste::insert(Liste*)
4116      1.2690  Polynome2::setStatus(int)
```

DEMO oprofile