

Efficient Use of R

Bjørn-Helge Mevik

Research Infrastructure Services Group, USIT, UiO

RIS Course Week

General Tips and Tricks

Speed Optimisation Procedure

Case Study: t Tests

General Points

- ▶ Use the latest possible R. Improvements happen all the time.
- ▶ Use a fast BLAS/LAPACK. On Abel we use Intel MKL.
- ▶ Use vector and matrix operations instead of loops. Example:

```
> xorig <- rnorm(1000000); x <- xorig
> system.time( x <- abs(x) )
      user  system elapsed
0.110    0.005    0.118
> x <- xorig
> system.time( for (i in seq_along(x)) x[i] <- abs(x[i]) )
      user  system elapsed
3.789    0.019    3.810
```

General Points, *cont.*

- ▶ Move calculations out of loops if possible.
- ▶ Pre-allocate long vectors and lists.
- ▶ Avoid frequent updates of data frames (use matrices instead).
Updating data frames is slow, and leads to more temporary copies.
- ▶ Use sparse matrices when possible. They can save both space and time.

General Points, *cont.*

- ▶ Avoid inverting matrices. Most calculations involving the inverse of a matrix can be done without actually calculating the inverse.
For instance: $L^{-1}M$ can be calculated as `solve(L, M)`.
- ▶ Similarly, avoid calculating the determinant of matrices. It is costly, and seldom needed.
- ▶ Use R's speed enhanced functions, like `colSums()`, `colMeans()`, `rowSums()`, `rowMeans()`, `crossprod()` and `tcrossprod()`. There are also a couple of fast versions of `seq()`; see `?seq`.

“Dangerous” Tips

These tips lead to complex code that is difficult to read, debug and maintain. Use with caution.

- ▶ Use underlying compute function directly if you call very general functions many times. For instance: `lm()` uses `lm.fit()`.
- ▶ If there is no underlying compute function, consider creating a stripped down version of the general function.
- ▶ Some R functions focus more on generality than speed. Consider writing a separate function for the special case at hand.
- ▶ If it is hard to improve the speed of a loop or function, it might be possible to rewrite it in C or Fortran.

Example: XL^{-1}

Q is a sparse matrix. XL^{-1} where $L = \text{chol}(Q)$ was calculated like this:

```
X <- matrix(rnorm(p * q), nrow = p, ncol = q)
t.solveL <- t(solve(chol(Q)))
for(i in 1:p){
  X[i,] <- as.numeric(t.solveL %*% X[i,])
}
```

Using the fact that $XL^{-1} = ((L^{-1})^T X^T)^T$, a better version is

```
tL <- t(chol(Q))
X <- as.matrix(t(solve(tL, matrix(rnorm(p * q), ncol = p))))
```

- ▶ It avoids calculating the inverse of `chol(Q)`,
- ▶ It avoids the `for` loop,
- ▶ It uses an efficient `Matrix` method for `solve`,
- ▶ Result: Time reduced from 80 to 0.05 seconds.

Speed Optimisation Procedure: Overview

Warning: Never optimise code before it works.

Steps:

1. Run the code manually.
2. Time the parts identified in the previous step.
3. Profile the code.
4. Optimise the code where needed and possible.
5. Try parallelising calculations where possible.
6. Try rewriting hot-spots in C or Fortran.

Optimisation Steps

1. **Run the code manually:** Check that it works, and find out which parts to focus on.
2. **Time the parts identified in the previous step.** Use realistic data if possible. Check how time scales wrt. data size and parameter values.

Tools:

- ▶ `system.time(ignore <- somefunc(X))`
- ▶ `rProf()` and `summaryRprof()`
- ▶ `microbenchmark` package

microbenchmark Example

For testing pre-allocation of vectors and list, I did:

```
microbenchmark(
  vec_alloc = { tmp <- numeric(len);
               for (i in seq_len(len)) { tmp[i] <- 1 } },
  vec_grow  = { tmp <- numeric(0);
               for (i in seq_len(len)) { tmp[i] <- 1 } },
  times = 5, unit = "s")
```

for different values of `len`, which gave outputs like:

Unit: seconds

	expr	min	lq	median	uq	max	neval
vec_alloc		0.028332	0.028942	0.028959	0.029649	0.031785	5
vec_grow		1.130030	1.148045	1.157901	1.167721	1.313788	5

Optimisation Steps *cont.*

3. Profile the code.

```
> Rprof(gc.profiling = TRUE, line.profiling = TRUE)
> bob <- make.Q(par, n.snps = 10000)
> Rprof(NULL); summaryRprof()
$by.self
      self.time self.pct total.time total.pct
"make.Q"      6.98   53.12      13.14   100.00
"col"         1.18    8.98       1.36    10.35
"=="         1.16    8.83       1.24     9.44
"row"         1.14    8.68       1.30     9.89 [...]
$by.total
      total.time total.pct self.time self.pct
"make.Q"      13.14   100.00      6.98   53.12
"col"         1.36    10.35       1.18    8.98
"row"         1.30     9.89       1.14    8.68
"=="         1.24     9.44       1.16    8.83 [...]
$sample.interval
[1] 0.02
$sampling.time
[1] 13.14
```

Optimisation Steps *cont.*

Adding line-based timings:

```
> summaryRprof(lines = "both")
[...]
```

	total.time	total.pct	self.time	self.pct
"make.Q"	13.14	100.00	0.00	0.00
bhm_sim_foo.R#35	7.88	59.97	4.64	35.31
bhm_sim_foo.R#37	4.14	31.51	2.34	17.81
"col"	1.36	10.35	1.18	8.98
"row"	1.30	9.89	1.14	8.68
"=="	1.24	9.44	1.16	8.83

```
[...]
```

```
$sample.interval
[1] 0.02
```

```
$sampling.time
[1] 13.14
```

Optimisation Steps *cont.*

4. **Optimise the code**, and benchmark the new code versus the old, for instance using `microbenchmark()`.
R's `debug()` is useful to step through the code, examine values, etc. Iterate from step 2 or 3. If more speedup is needed, continue with step 5.
5. **Parallelise calculations where possible**. Several options, e.g.: Run several R jobs (`arrayrun`), use packages like `parallel`, use GPUs with packages like `gputools`.
6. **Try rewriting hot-spots in C or Fortran**. Probably most useful if the section of code is doing a lot of manipulations or element-wise calculations. Popular tools:
 - ▶ The `Rcpp` package
 - ▶ The `inline` package

Case Study: t Tests

- ▶ Illustrates steps 1 – 4 of the optimisation procedure.
- ▶ Task: run t tests on the columns of a matrix with 2,000,000 columns.
- ▶ Straightforward implementation:

```
n <- 1000; m <- 2000000  
X <- matrix(rnorm(n*m), ncol = m)  
res <- apply(X, 2, t.test)
```

t Tests *cont.*

1. Running code with $m = 50000$ shows all time is spent inside `apply()`.
2. Timing code with different values of m :

Coloumns	Seconds
25,000	4,36
50,000	9,74
100,000	21,14
2,000,000	528,44

i.e., approximately linear in m .

t Tests *cont.*3 & 4 Profiling with `m = 100000`:

```

> Rprof(gc.profiling = TRUE, line.profiling = TRUE)
> res <- apply(X, 2, t.test)
> Rprof(NULL); summaryRprof()
[...]
```

\$by.total	total.time	total.pct	self.time	self.pct
"apply"	21.52	100.00	0.08	0.37
"FUN"	20.68	96.10	1.84	8.55
"t.test.default"	18.54	86.15	3.90	18.12
"deparse"	6.36	29.55	1.22	5.67
"match.arg"	3.90	18.12	0.30	1.39
"<GC>"	3.82	17.75	3.82	17.75
"eval"	3.06	14.22	0.32	1.49
"%in%"	2.74	12.73	0.26	1.21
"var"	2.62	12.17	0.30	1.39
".deparse0pts"	2.58	11.99	0.54	2.51
"match"	2.54	11.80	0.74	3.44

```

[...]
```

t Tests *cont.*

- ▶ Sampling time was 21.52 seconds
- ▶ 95 % of the time is spent in `t.test()` (“FUN”),
- ▶ 85 % inside `t.test.default()`.
- ▶ Except for `var()`, the time is spent in “administrative” functions like `deparse()`, `match.arg()` and `eval()`
- ▶ Let us try a bare-bones implementation of `t.test()`

(Show original functions.)

t Tests *cont.*

A stripped version of this could look like

```
t.test.minimal <- function(x) {
  mu <- 0; conf.level <- 0.95
  nx <- length(x)
  mx <- mean(x)
  vx <- var(x)
  df <- nx - 1
  stderr <- sqrt(vx/nx)
  tstat <- (mx - mu)/stderr
  estimate <- mx
  pval <- 2 * pt(-abs(tstat), df)
  alpha <- 1 - conf.level
  cint <- qt(1 - alpha/2, df)
  cint <- tstat + c(-cint, cint)
  cint <- mu + cint * stderr
  rval <- list(statistic = tstat, parameter = df, p.value = pval,
              conf.int = cint, estimate = estimate, null.value = mu)
  return(rval)
}
```

t Tests *cont.*

Profiling this with 100.000 coloumns:

```
> Rprof(gc.profiling = TRUE, line.profiling = TRUE)
> res <- apply(X, 2, t.test.minimal)
> Rprof(NULL); summaryRprof()
[...]
```

\$by.total

	total.time	total.pct	self.time	self.pct
"apply"	7.18	100.00	0.12	1.67
"FUN"	6.30	87.74	1.64	22.84
"var"	2.24	31.20	0.30	4.18
"mean"	1.18	16.43	0.70	9.75
"stopifnot"	0.98	13.65	0.30	4.18
"aperm.default"	0.76	10.58	0.76	10.58
"aperm"	0.76	10.58	0.00	0.00
"qt"	0.52	7.24	0.28	3.90
"match.call"	0.50	6.96	0.26	3.62
".Call"	0.46	6.41	0.44	6.13
"mean.default"	0.46	6.41	0.44	6.13
"pt"	0.38	5.29	0.20	2.79
[...]				

t Tests *cont.*

- ▶ Sampling time was 7.18 seconds, i.e., 3 times as short
- ▶ 30 % of the time is spent in `var()`
- ▶ 15 % inside `mean()`.
- ▶ Still some administrative overhead
- ▶ Calculations are simple; let us try vectorising it

t Tests *cont.*

Vectorised version:

```
t.test.matrix <- function(X) {  
  mu <- 0; conf.level <- 0.95  
  nx <- nrow(X)  
  mx <- colMeans(X)  
  vx <- apply(X, 2, var)  
  df <- nx - 1  
  stderr <- sqrt(vx/nx)  
  tstat <- (mx - mu)/stderr  
  estimate <- mx  
  pval <- 2 * pt(-abs(tstat), df)  
  alpha <- 1 - conf.level  
  rval <- list(statistic = tstat, parameter = df, p.value = pval,  
    estimate = estimate, null.value = mu)  
  return(rval)  
}
```

t Tests *cont.*

Profiling it:

```
> Rprof(gc.profiling = TRUE, line.profiling = TRUE)
```

```
> res <- t.test.matrix(X)
```

```
> Rprof(NULL); summaryRprof()
```

```
[...]
```

```
$by.total
```

	total.time	total.pct	self.time	self.pct
"t.test.matrix"	4.10	100.00	0.00	0.00
"apply"	3.98	97.07	0.12	2.93
"FUN"	3.06	74.63	0.20	4.88
"is.data.frame"	1.14	27.80	1.14	27.80
"stopifnot"	0.98	23.90	0.40	9.76
"aperm.default"	0.76	18.54	0.76	18.54
"aperm"	0.76	18.54	0.00	0.00
".Call"	0.52	12.68	0.52	12.68
"match.call"	0.42	10.24	0.28	6.83
"pmatch"	0.22	5.37	0.20	4.88
"sys.call"	0.14	3.41	0.08	1.95
"colMeans"	0.10	2.44	0.10	2.44
[...]				

t Tests *cont.*

- ▶ Sampling time 4.1 seconds, i.e. five times as fast as the original.
- ▶ 2,000,000 columns now takes about 100 seconds, so we are perhaps “flogging a dead horse”, but let’s see how far we can push it
- ▶ Now most time spent calculating the variance (95 % in `apply()` and 75 % in `var()`)
- ▶ Some overhead inside `var()` could possibly be removed. Let’s try.

t Tests *cont.*

`var()` looks like this:

```
function (x, y = NULL, na.rm = FALSE, use)
{
  if (missing(use))
    use <- if (na.rm)
      "na.or.complete"
    else "everything"
  na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise",
    "everything", "na.or.complete"))
  if (is.na(na.method))
    stop("invalid 'use' argument")
  if (is.data.frame(x))
    x <- as.matrix(x)
  else stopifnot(is.atomic(x))
  if (is.data.frame(y))
    y <- as.matrix(y)
  else stopifnot(is.atomic(y))
  .Call(C_cov, x, y, na.method, FALSE)
}
```

t Tests *cont.*

An bare-bones version would be:

```
var.minimal <- function(x) {
  ## "4" is the na.method "everything"
  .Call(stats:::C_cov, x, NULL, 4, FALSE)
}
```

(*C_cov* is not exported, therefore *stats:::C_cov*.) Timing this against the default default *var()* shows that it is 25 % faster:

```
> microbenchmark(
+   var      = { apply(X, 2, var) },
+   var_min  = { apply(X, 2, var.minimal) },
+   times = 5, unit = "s")
```

Unit: seconds

expr	min	lq	median	uq	max	neval
var	3.994110	4.134170	4.146463	4.193262	4.323972	5
var_min	2.991741	3.013115	3.053666	3.103311	3.112358	5

t Tests *cont.*

Using this instead of `var` in our *t* test function (code not shown) gives:

```
> Rprof(gc.profiling = TRUE, line.profiling = TRUE)
> res <- t.test.matrix.minvar(X)
> Rprof(NULL); summaryRprof()
[...]
```

	total.time	total.pct	self.time	self.pct
"t.test.matrix.minvar"	3.12	100.00	0.00	0.00
"apply"	2.98	95.51	0.08	2.56
"FUN"	2.10	67.31	1.16	37.18
"aperm.default"	0.76	24.36	0.76	24.36
"aperm"	0.76	24.36	0.00	0.00
":::"	0.48	15.38	0.10	3.21
".Call"	0.46	14.74	0.46	14.74
"get"	0.32	10.26	0.10	3.21
"asNamespace"	0.22	7.05	0.06	1.92
"colMeans"	0.10	3.21	0.10	3.21

```
[...]
$sampling.time
[1] 3.12
```

t Tests *cont.*

- ▶ Sampling time 3.12 seconds, so 25 % faster than before.
- ▶ Some overhead in `apply()`, so let's try a vectorised `var()`:

```
var.matrix <- function(X) {
  nx <- nrow(X)
  mx <- colMeans(X)
  (X - rep(mx, each = nx))^2 / (nx - 1) #Faster than sweep(X, 2, mx)
}
> microbenchmark(
+   var_min = { apply(X, 2, var.minimal) },
+   var_mat = { var.matrix(X) },
+   times = 5, unit = "s")
Unit: seconds
      expr      min       lq    median       uq      max neval
var_min 2.998382 3.018809 3.024215 3.034598 3.084568     5
var_mat 3.050913 3.051260 3.055055 3.056435 3.067346     5
```

Not faster than `var.minimal()`, so we drop it. It is hard to beat C!

t Tests *cont.*

The optimised function scales linearly with the number of columns:

