Algorithm for ancestry analysis and its performance optimization

Beilei Bian

2016.12.11@Shanghai

Outline

- What is ancestry analysis?
- Statistical Model
- Algorithms for optimization
- Performant code

What is ancestry analysis?

Tracing your ancestry

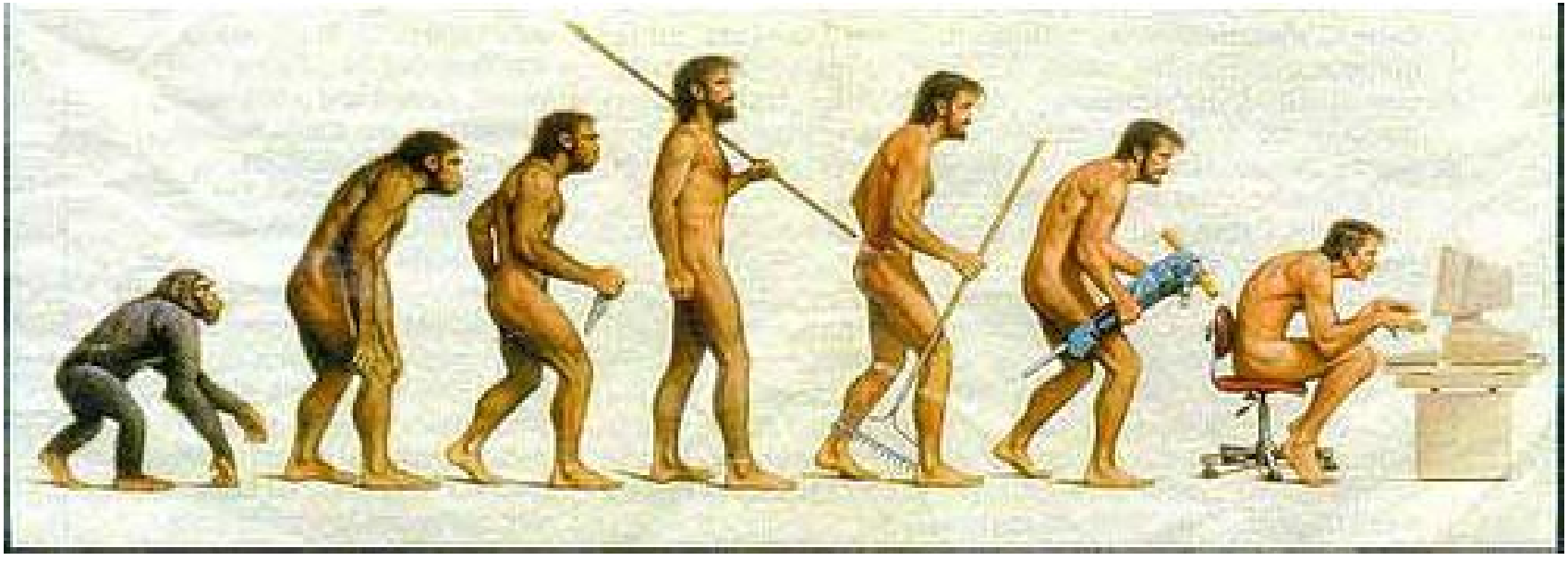

Figure 1: Humans never stop evolving

What is ancestry analysis?

- Estimate ancestries from large SNP genotype data

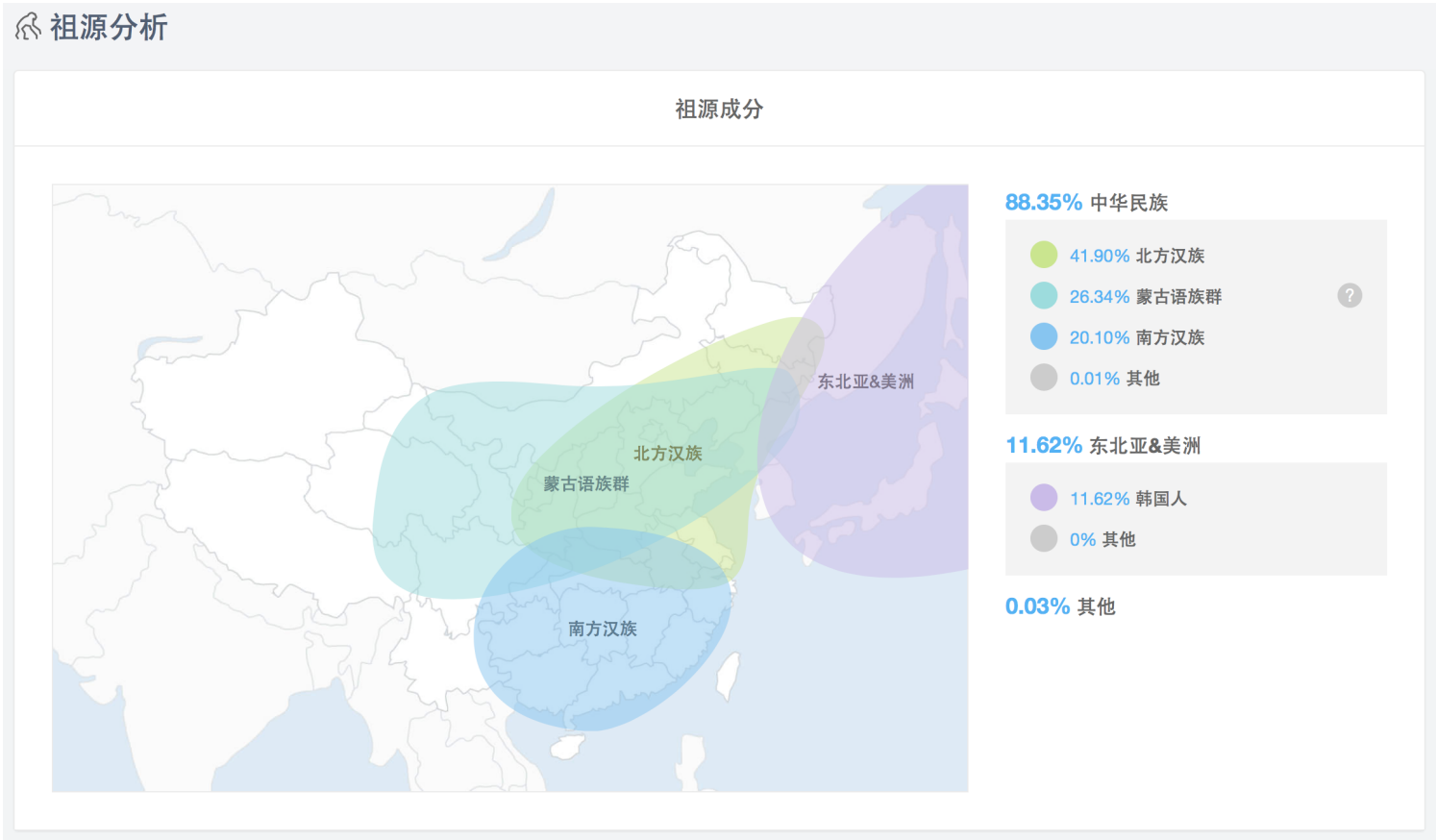

Figure 2: Ancestry result - WeGene

Statistical Model

- Genotype Matrix

Individual ID	g1	g2	g3
1	0	2	0
2	0	0	0
3	1	0	1
4	2	0	1

- Ancestry Matrix

Individual ID	CEU	YRI	French
1	0.99998	0.00001	0.00001
2	0.00001	0.99998	0.00001
3	0.00001	0.00001	0.99998
4	Unknown	Unknown	Unknown

Statistical Model

- Frequency Matrix

Population	SNP1	SNP2	SNP3
CEU	0.9	0.2	0.1
YRI	0.5	0.4	0.2
French	0.01	0.05	0.1

Statistical Model

- Binomial distribution

$$
\begin{aligned}
\Pr(1/1 \text{ for i at SNP j}) &= \Big[\sum_k q_{ik} f_{kj}\Big]^2 \\
\Pr(1/2 \text{ for i at SNP j}) &= 2\Big[\sum_k q_{ik} f_{kj}\Big]\Big[\sum_k q_{ik}(1-f_{kj})\Big] \qquad (1) \\
\Pr(2/2 \text{ for i at SNP j}) &= \Big[\sum_k q_{ik}(1-f_{kj})\Big]^2.
\end{aligned}
$$

Figure 3: Binomial distribution

Loglikelihood function

$L(Q,F) = \sum_i \sum_j \{g_{ij} ln[\sum_k q_{ik} f_{kj}] + (2-g_{ij}) ln[\sum_k q_{ik}(1-f_{kj})]\}$

Algorithms for optimization

- EM

EM is numerically more stable, however, the stability of EM is attained at the expense of slow, linear convergence.

- Block Relaxation

The general idea of is easy to understand. It minimizes a real-valued function of several variables by partitioning the variables into blocks. And we choose initial values for all blocks, and then minimize over one of the blocks, while keeping all other blocks fixed at their current values.

`SQP: Sequential Quadratic Programming`

- Quasi-Newton Acceleration

The EM algorithm of FRAPPE updates the parameters via

$$f_{kj}^{n+1} = \frac{\sum_i g_{ij} a_{ijk}^n}{\sum_i g_{ij} a_{ijk}^n + \sum_i (2 - g_{ij}) b_{ijk}^n}, \tag{3}$$

$$q_{ik}^{n+1} = \frac{1}{2J} \sum_j \left[g_{ij} a_{ijk}^n + (2 - g_{ij}) b_{ijk}^n \right], \tag{4}$$

where for convenience we define

$$a_{ijk}^n = \frac{q_{ik}^n f_{kj}^n}{\sum_m q_{im}^n f_{mj}^n}, \quad b_{ijk}^n = \frac{q_{ik}^n (1 - f_{kj}^n)}{\sum_m q_{im}^n (1 - f_{mj}^n)}.$$

Figure 4: EM

separately. The entries of the first differentials are

$$\frac{\partial L}{\partial q_{ik}} = \sum_j \left[\frac{g_{ij} f_{kj}}{\sum_m q_{im} f_{mj}} + \frac{(2 - g_{ij})(1 - f_{kj})}{\sum_m q_{im}(1 - f_{mj})} \right],$$

$$\frac{\partial L}{\partial f_{kj}} = \sum_i \left[\frac{g_{ij} q_{ik}}{\sum_m q_{im} f_{mj}} - \frac{(2 - g_{ij}) q_{ik}}{\sum_m q_{im}(1 - f_{mj})} \right].$$

All entries of the second differentials vanish except for

$$\frac{\partial^2 L}{\partial q_{ik} q_{il}} = -\sum_j \left\{ \frac{g_{ij} f_{kj} f_{lj}}{\left(\sum_m q_{im} f_{mj}\right)^2} + \frac{(2 - g_{ij})(1 - f_{kj})(1 - f_{lj})}{\left[\sum_m q_{im}(1 - f_{mj})\right]^2} \right\},$$

$$\frac{\partial^2 L}{\partial f_{kj} \partial f_{lj}} = -\sum_i \left\{ \frac{g_{ij} q_{ik} q_{il}}{\left(\sum_m q_{im} f_{mj}\right)^2} + \frac{(2 - g_{ij}) q_{ik} q_{il}}{\left[\sum_m q_{im}(1 - f_{mj})\right]^2} \right\},$$

Figure 5: differentials

Version 1 (EM)

```
updatef1 <- function(g, q, f) {
    f1 <- matrix(NA, ncol(q), ncol(f)); a <- rep(NA, nrow(q))
    b <- rep(NA, nrow(q))
    for(k in 1:ncol(q)){
        for(j in 1:ncol(f)){
            for(i in 1:nrow(q)){
                a[i] <- g[i, j] * q[i, k] * f[k, j] /
                        q[i, ] %*% f[, j]
                b[i] <- (2 - g[i, j]) * q[i, k] * (1 - f[k, j])
                        q[i, ] %*% (1 - f[, j])
            }
            f1[k, j] <- sum(a) / (sum(a) + sum(b))
        }
    }
    return(f1)
}
```

Version 1 (second differential)

```
loglikhessf <- function(g, q, f, s) {
  hessf <- rep(NA, n); K <- ncol(q)
  loglikhessvaluef <- matrix(NA, K, K)
  for(k in 1:K){
    for(l in 1:K){
      for(i in 1:n){
        hessf[i]<- -(g[i, s] * q[i, k] * q[i, l] /
                          (q[i, ] %*% f[, s])^2) +
                      ((2 - g[i, s]) * q[i, k] * q[i, l]) /
              (q[i, ] %*% (1 - f[, s]))^2
      }
      loglikhessvaluef[k, l] <- sum(hessf)
    }
  }
  return(loglikhessvaluef)
}
```

How to evaluate and opimize the programs?

Performant code

- Steps
  1. Try to find the slowest part of the code. – Measuring performance
  2. Improving performance

How to measure the performance of the code?

First, you need a profiler. R uses a sampling profiler which is simple and lightweight.

Tool: Profvis

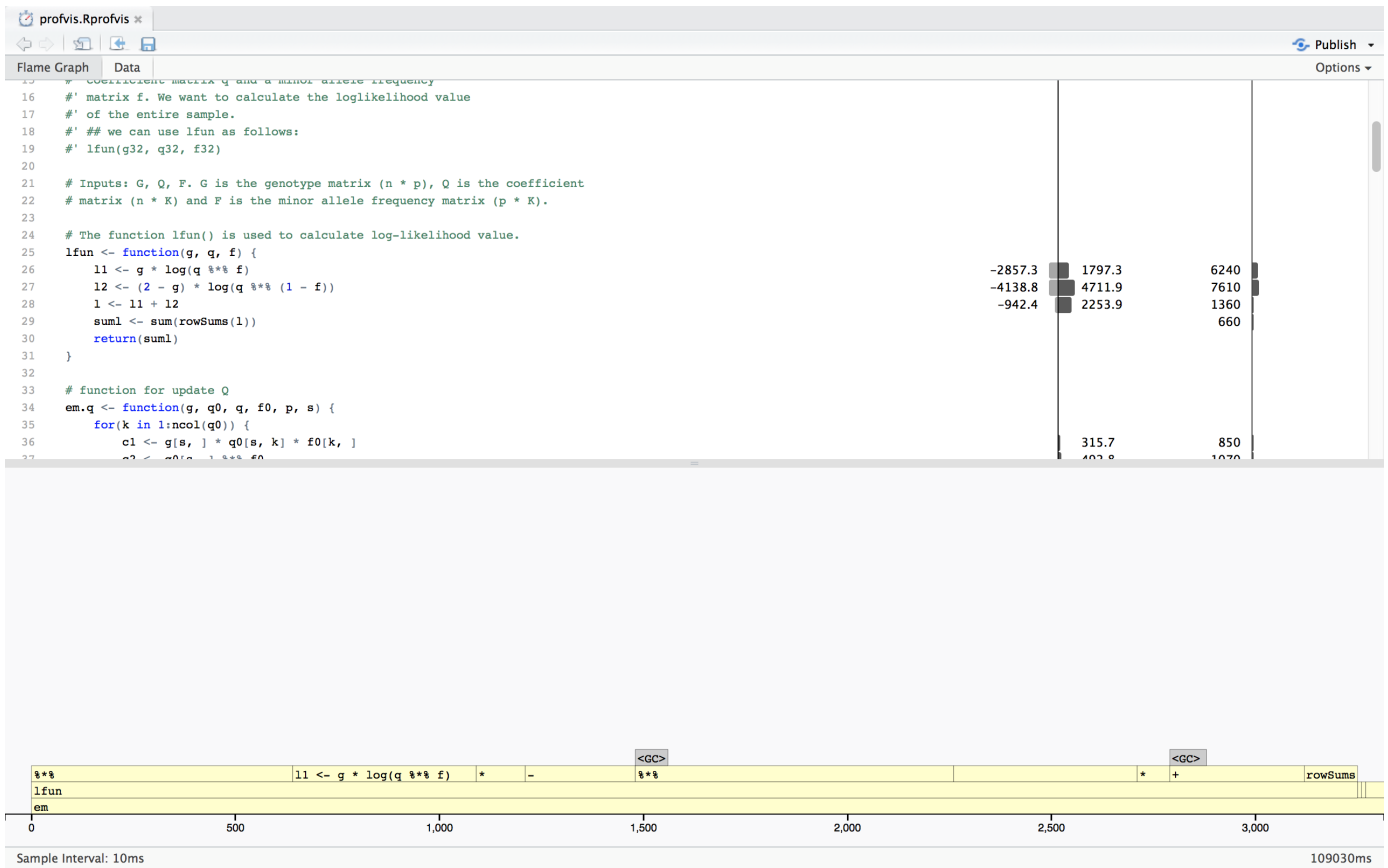

Figure 6: Profvis

Tool: Profvis

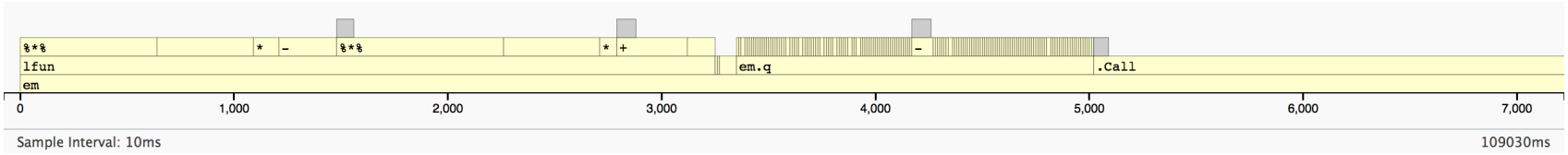

Figure 7: flame graph

horizontal direction: time

vertical direction: call stack

- Advantages:
  - interactive
  - memory & time
- Limitations:
  - cannot show up internal R functions and code implemented by other languages(C, C++, Fortran)

Other tools for profiling R code

- package
  - microbenchmark
  - proftools
  - GUIProfiler
  - aprof
- function
  - system.time()
- Advanced profiling
  - pbdPAPI: **hardware performance counters like cache misses, flops**

pbdR: programming with big data in R

Improving performance

```
updatef1 <- function(g, q, f) {
    f1 <- matrix(NA, ncol(q), ncol(f)); a <- rep(NA, nrow(q))
    b <- rep(NA, nrow(q))
    for(k in 1:ncol(q)){
        for(j in 1:ncol(f)){
            for(i in 1:nrow(q)){
                a[i] <- g[i, j] * q[i, k] * f[k, j] /
                        q[i, ] %*% f[, j]
                b[i] <- (2 - g[i, j]) * q[i, k] * (1 - f[k, j])
                        q[i, ] %*% (1 - f[, j])
            }
            f1[k, j] <- sum(a) / (sum(a) + sum(b))
        }
    }
    return(f1)
}
```

Improving performance

- Vectorize

```
updatef2 <- function(g, q, f) {
    f1 <- matrix(NA, ncol(q), ncol(f))
    for(k in 1:ncol(q)) {
        for(j in 1:ncol(f)) {
            a <- (g[, j] * q[, k] * f[k, j]) /
                 (q %*% f[, j])
            b <- ((2 - g[, j]) * q[, k] * (1 - f[k, j])) /
                 (q %*% (1 - f[, j]))
            f1[k, j] <- sum(a) / (sum(a) + sum(b))
        }
    }
    return(f1)
}
```

microbenchmark

```
library(microbenchmark)
microbenchmark(
    uf1 <- updatef1(g3, q3, f3),
    uf2 <- updatef2(g3, q3, f3),
    times = 3
)

#> Unit: milliseconds
#>                          expr         min          lq       mean
#> uf1 <- updatef1(g3, q3, f3) 16901.4165 17107.2004 17387.713 1
#> uf2 <- updatef2(g3, q3, f3)   591.9372   598.6475   604.927
#>         max neval
#>17948.7376     3
#>  617.4861     3
```

- R'C Interface

Suppose you've done everything you can in R, but your code still isn't fast enough.

- Why is R slow?
  - Interpreted language
  - Extreme dynamism, but complicate the implementor's task
  - ...
- Use compiled language: **R'C API/Rcpp**
  - fast execution
  - editing and debugging is slower than interpreted language

.Call()

- Example

```
// calculate sum of first 100 natural number
// include the header files
# include <R.h>
# include <Rinternals.h>
# include <Rmath.h>
SEXP hello(SEXP vec) {
    int n = length(vec);
    double *vec1, *a;
    vec1 = REAL(vec);
    SEXP out = PROTECT(allocVector(REALSXP, 1));
    a = REAL(out);
    for(int i = 0; i < n; i++) {
        a[0] += vec1[i];
    }
    UNPROTECT(1);
    return(out);
}
```

Comparison

```
microbenchmark(
    uf1 <- updatef1(g3, q3, f3),
    uf2 <- updatef2(g3, q3, f3),
    uf3 <- .Call("updatef", nrow(q3), ncol(g3), ncol(q3),
                 q3, f3, g3),
    times = 3
)

#>                          expr
#> uf1 <- updatef1(g3, q3, f3)
#> uf2 <- updatef2(g3, q3, f3)
#> uf3 <- .Call("updatef", nrow(q3), ncol(g3), ncol(q3),
#>               q3, f3, g3)
#>         mean  neval
#>  16862.84830  3
#>    549.80406  3
#>     26.00166  3
```

Other tricks for matrix calculation

- **crossprod(x, y)**

```
## Unit: milliseconds
##                          expr       min         lq       mean     m
##       t1 <- t(g32) %*% q32 1310.0065 1322.1729 1645.5548 1332
##  t2 <- crossprod(g32, q32)  477.9439  504.1984  509.2398  516
##         uq       max neval
##  1358.4614 2904.9669     5
##   520.4295  527.1612     5
```

- **rowSums, colSums**

```
library(radmixture)
lfun1 <- function(g, q, f) {
    l1 <- g * log(q %*% f)
    l2 <- (2 - g) * log(q %*% (1 - f))
    l <- l1 + l2
    suml <- sum(rowSums(l))
    return(suml)
}
lfun2 <- function(g, q, f) {
    l1 <- g * log(q %*% f)
    l2 <- (2 - g) * log(q %*% (1 - f))
    l <- l1 + l2
    suml <- sum(apply(l, 1, FUN = sum))
    return(suml)
}
#> Unit:seconds
#> expr1: 2.636637
#> expr2: 6.587903
```

- **use OpenBLAS/MKL**

Ubuntu: **sudo apt-get libopenblas-base**

sudo update-alternatives –config libblas.so.3

```
  Selection    Path                                      Priority   Status
------------------------------------------------------------
* 0            /usr/lib/openblas-base/libblas.so.3        40        auto mode
  1            /usr/lib/libblas/libblas.so.3              10        manual mode
  2            /usr/lib/openblas-base/libblas.so.3        40        manual mode
```

Figure 8: update the link

- **use OpenBLAS**

libblas

```
> microbenchmark(lfun(g32,q32,f32),times=5)
Unit: seconds
                expr      min       lq     mean  median       uq      max neva
l
 lfun(g32, q32, f32) 11.57285 11.60384 12.17303 11.7427 11.76221 14.18355
5
```

Figure 9: blas

openblas-base

```
> microbenchmark(lfun(g32,q32,f32),times=5)
Unit: seconds
                expr      min       lq     mean   median       uq      max
 lfun(g32, q32, f32) 7.928497 8.096206 8.168326 8.145867 8.269681 8.401381
 neval
     5
```

Figure 10: openblas

Parallelise

task-parallelism and data-parallelism

- parallel
- foreach

```
library(foreach)
library(doMC)
registerDoMC(cores = 10)
flongvec <- foreach(j = 1:ncol(f), .combine = cbind,
                    .packages = c("radmixture", "quadprog"))
            %dopar% {
                bvec <- c(-f[, j], f[, j] - 1)
                flongvec[, j] <- f[, j] +
                                 solve.QP(-diff.f(g, q, f, s = j)
                                 diff.f(g, q, f, s = j)$Jaco,
                                 t(Amatf), bvec)$solution
            }
```

Other techniques

- Avoid copy :**R makes a copy of f to a new location**

```
library(pryr)
c(address(f), refs(f))
#> [1] "0x127d2b000" "1"
f[, j] <- f[, j] + solve.QP(-diff.f(g, q, f, s = j)$Hess,
                            diff.f(g, q, f, s = j)$Jaco,
                            t(Amatf), bvec)$solution
f <- 1:10
ff <- f
c(address(f), address(ff), refs(ff))
#> [1] "0x129a56000" "0x129a56000" "2"
f[5] <- 10
c(address(f), address(ff))
#> [1] "0x11294ea08" "0x129a56000"
```

- Avoid copy

```
fnew <- matrix(NA, nrow(f), ncol(f))
fnew[, j] <- f[, j] + solve.QP(-diff.f(g, q, f, s = j)$Hess,
                               diff.f(g, q, f, s = j)$Jaco,
                               t(Amatf), bvec)$solution
```

expr1: 2857000ms

expr2: 316470ms

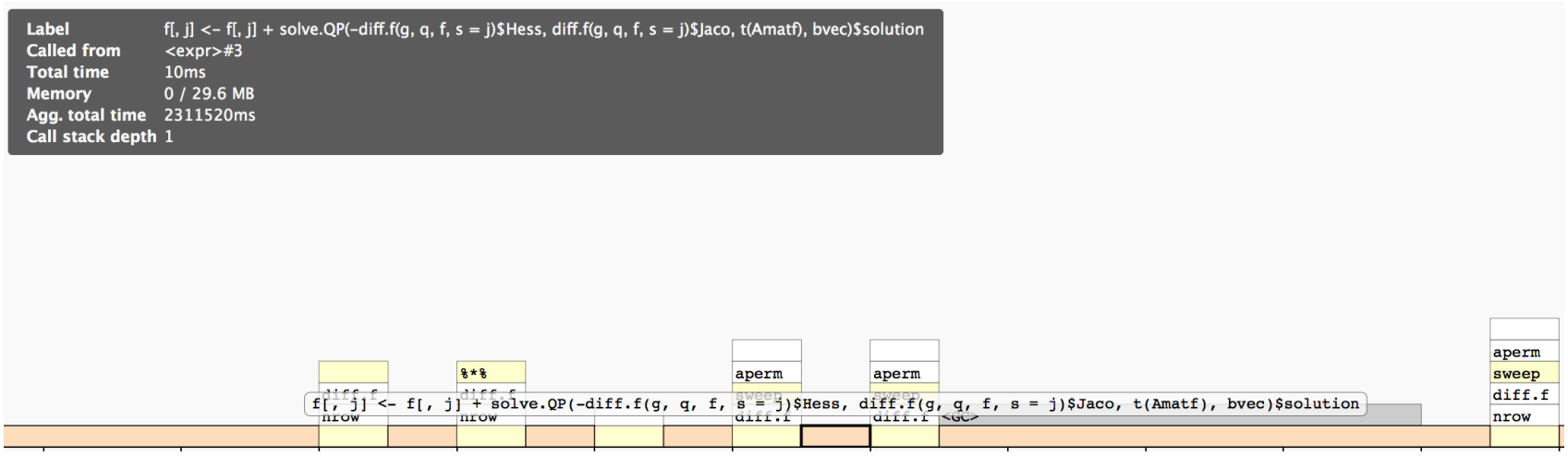

Figure 11: slowness because of modifying a copy

- Byte code compilation **compiler**

```
lfun_c <- compiler::cmpfun(lfun)
microbenchmark::microbenchmark(ll1 <- lfun(g32, q32, f32),
                               ll2 <- lfun_c(g32, q32, f32),
                               times = 3)
```

```
## Unit: seconds
##                              expr      min       lq     mean   me
##    ll1 <- lfun(g32, q32, f32) 2.635201 2.745057 2.818596 2.85
##  ll2 <- lfun_c(g32, q32, f32) 2.591420 2.604328 2.856135 2.61
##       max neval
##  2.965672     3
##  3.359748     3
```

But in this example, byte code compilation doesn't help at all.

In DISCRIPTION: ByteCompile: true

In python, you can use Cython.

- file I/O
  - read.table() – slow

```
system.time(f <- read.table(file = "/Users/bianbeilei/f.txt"
#> user  system elapsed
#> 7.830   0.063   7.919
```

  - fread() and fwrite() – fast

```
library(data.table)
system.time(f <- fread("/Users/bianbeilei/f.txt"))
#> user  system elapsed
#> 0.269   0.002   0.271
```

  - readLines() – fastest(string type)

```
system.time(f <- readLines("/Users/bianbeilei/f.txt"))
#> user  system elapsed
#> 0.049   0.001   0.049
```

- Alternative R implementations: qpR, Renjin, FastR
- Use GPU
- bookdown.org: Efficient R programming

References

[1] D.H. Alexander, J. Novembre, and K. Lange. Fast model-based estimation of ancestry in unrelated individuals. Genome Research, 19:1655–1664, 2009.

[2] H. Zhou, D. H. Alexander, and K. Lange. A quasi-Newton method for accelerating the convergence of iterative optimization algorithms. Statistics and Computing, 2009.

[3] Advanced R by Hadley Wickham

[4] Lim A, Tjhi W. R High Performance Programming[M].

Contact me: bblzuiaimessi@gmail.com/bianbeilei@wegene.com