Benchmarking gradient functions

Ben Bolker

April 21, 2015

```
library("fitsir")
library("deSolve")
library("Rcpp")
library("ggplot2"); theme_set(theme_bw())
library("microbenchmark")
```

The built-in gradient function is `SIR.grad`:

```
SIR.grad

## function(t, x, params) {
##     g <- with(as.list(c(x,params)),
##           {
##               c(-beta*exp(logI)*S/N,beta*S/N-gamma)
##           })
##     list(g)
## }
```

Does using `with()` incur a performance cost?

```
SIR.grad2 <- function(t, y, params) {
    list(c(-params[1]*exp(y[2])*y[1]/params[3],
            params[1]*y[1]/params[3]-params[2]))
}
## without division by N
SIR.grad4 <- function(t, y, params) {
    list(c(-params[1]*exp(y[2])*y[1],
            params[1]*y[1]-params[2]))
}
## define Jacobian, maybe useful later ...
jacfunc <- function(t, y, params) {
    matrix(c(-exp(y[2])*y[1],0,
             y[1],-1),nrow=2,byrow=TRUE)
}
```

Rcpp version:

```
sourceCpp("sirgrad.cpp")
```

C version:

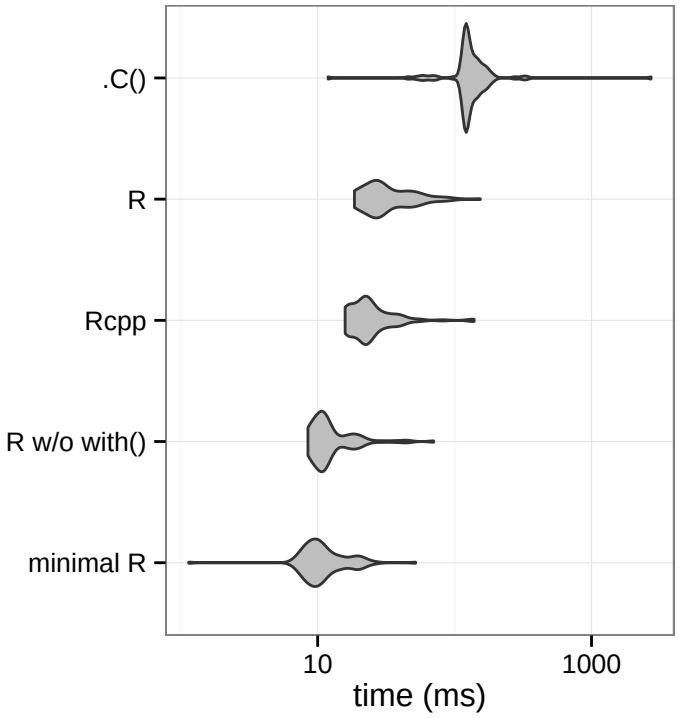
```
system("R CMD SHLIB sirgrad.c")
dyn.load(paste0("sirgrad",.Platform$dynlib.ext))
## avoid redefining grade vector every time:
## should be a little more careful and define
## a function closure/ put grad in the environment
## of SIR.grad3
grad <- numeric(2) ## special case
SIR.grad3 <- function(t, y, params) {
   .C("derivs0",tvec, start, grad, pars)[3]
}
```

```
start <- c(S=0.99,logI=log(0.01))
pars <- c(beta=2,gamma=1,N=1)
pars2 <- with(as.list(pars),c(beta=beta/N,gamma=gamma))
tvec <- seq(0,2000,by=0.1)
funList <- list(sirgrad,SIR.grad,
                SIR.grad2,SIR.grad3)
testfun <- function(f,P=pars) f(0,start,P)
do.call(rbind,sapply(funList,testfun))

##         beta beta
## grad -0.0198 0.98
##      -0.0198 0.98
##      -0.0198 0.98
##      -0.0198 0.98
```

```
m1 <- microbenchmark(testfun(sirgrad),
                     testfun(SIR.grad),
                     testfun(SIR.grad2),
                     testfun(SIR.grad3),
                     testfun(SIR.grad4,P=pars2))
levels(m1$expr) <- c("Rcpp",
                     "R",
                     "R w/o with()",
                     ".C()",
                     "minimal R")
m1$expr <- reorder(m1$expr,m1$time)
```

```
(b1plot <- ggplot(m1,aes(y=time/1e3,x=expr))+geom_violin(fill="gray")+
    scale_y_log10() +
        labs(x="",y="time (ms)")+
    coord_flip())
```



Conclusion: unless I've screwed something up, the advantage to coding in C stems from the ability to call the gradient function directly, *not* from computing the gradient function itself faster. Of course, this conclusion could change a lot with more complex gradient functions. The more complex the function, and the more it needs to use explicit looping constructs (rather than vectorized or matrix computations), the bigger the win from compiling the gradient function is likely to be.

For optimizing fits of functions based on solutions of some of the techniques and reference suggested for MATLAB, here and here, may help. More specifically, should look at Raue et al. (2013).

For example:

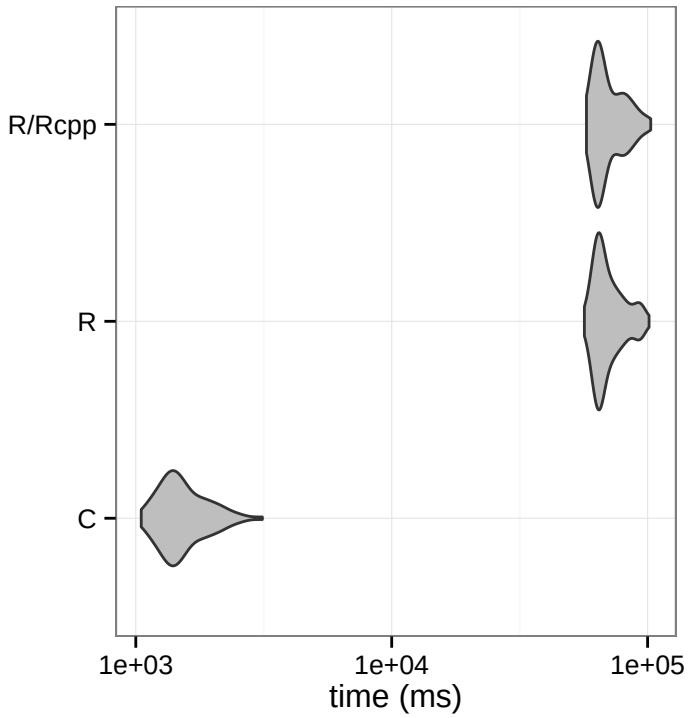
- use a fixed-stepsize solver;
- avoid optimization methods that compute finite differences;
- compute gradients of the objective function directly by solving auxiliary ODEs (sensitivity equations)

The last is possibly most interesting but most difficult.

```
pars3 <- c(pars,i0=0.01)
tvec <- seq(0,20,by=0.01)
runODE.C <- function(t, params) {
    odesol <- with(as.list(params),
                   ode(y=c(S=N,logI=log(N*i0)),
                       times=t,
                       func="derivs",
                       parms=params[1:3],
                       dllname = "sirgrad",
                       initfunc = "initmod",
                       nout = 1, outnames = character(0)))
    return(odesol[,"logI"])
}
runODE.R <- function(t, params, func=SIR.grad) {
    odesol <- with(as.list(params),
                   ode(times=t,
                       y=c(S=N,logI=log(N*i0)),
                       func=func,
                       parms=params[1:3]))
    return(odesol[,"logI"])
}
stopifnot(all.equal(runODE.R(tvec,pars3),runODE.C(tvec,pars3)))
```

```
m2 <- microbenchmark(runODE.R(tvec,pars3),
                     runODE.R(tvec,pars3,func=sirgrad),
                     runODE.C(tvec,pars3))
levels(m2$expr) <- c("R",
                     "R/Rcpp",
                     "C")
m2$expr <- reorder(m2$expr,m2$time)
```

Results replicate Irena Papst's, confirming that the performance issue (at least for this simple a gradient function) is the :



Is there a way to compile `Rcpp` to a DLL so it can be used via the wrapper?

References

Raue, A., M. Schilling, J. Bachmann, A. Matteson, M. Schelke, D. Kaschek, S. Hug, C. Kreutz, B. D. Harms, F. J. Theis, U. Klingmller, and J. Timmer (2013, September). Lessons learned from quantitative dynamical modeling in systems biology. *PLoS ONE 8*(9), e74335.