Benchmarking gradient functions

Ben Bolker

September 23, 2016

Load packages:

```r
library(fitsir)
library(deSolve)
if (packageVersion("odeintr")<"1.5")
    stop("need devel version: try devtools::github_install('thk686/odeintr')")
library(odeintr)
library(Rcpp)
library(ggplot2); theme_set(theme_bw())
library(microbenchmark)
```

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

```r
SIR.grad

## function (t, x, params)
## {
##     g <- with(as.list(c(x, params)), {
##         I = exp(logI)
##         dS = -beta * exp(logI) * S/N
##         dlogI = beta * S/N - gamma
##         list(c(dS, dlogI), I = I)
##     })
## }
## <environment: namespace:fitsir>
```

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

```r
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:

```
s <- sourceCpp("sirgrad.cpp")
```

```
dllfn <- list.files(file.path(s$buildDirectory),pattern="\\.so$")
ss <- system(sprintf("nm -g %s",file.path(s$buildDirectory,dllfn)),
             intern=TRUE)
rcppgradfn <- tail(strsplit(grep("sirgrad$",ss,value=TRUE)," ")[[1]],1)
rcppinitfn <- tail(strsplit(grep("initmod",ss,value=TRUE)," ")[[1]],1)
## sourceCpp_1_sirgrad
## already loaded ...
## dyn.load(file.path(s£buildDirectory,"sourceCpp_2.so"))
## test:
start <- c(S=0.99,logI=log(0.01))
pars <- c(beta=2,gamma=1,N=1)
.Call(rcppgradfn,0,start,pars)

## $grad
## [1] -0.0198  0.9800
```

C version:

```
system("R CMD SHLIB sirgrad.c")
dyn.load(paste0("sirgrad",.Platform$dynlib.ext))
## avoid redefining grad 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]
}
```

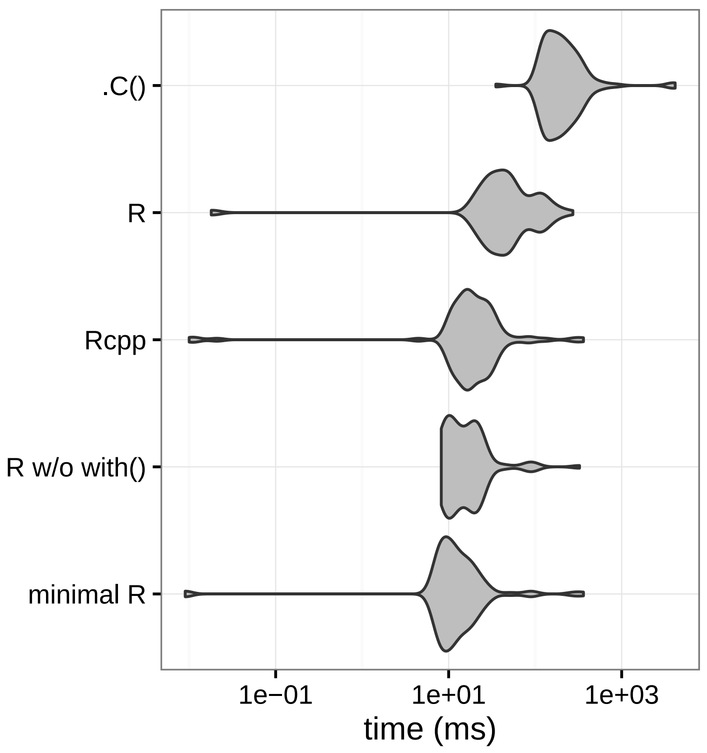
```
start <- c(S=0.99,logI=log(0.01))
pars <- c(beta=2,gamma=1,N=1)
```

```
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))

##                  I
## [1,] Numeric,2 Numeric,2
## [2,] Numeric,2 0.01
## [3,] Numeric,2 Numeric,2
## [4,] Numeric,2 Numeric,2
```

```
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.

We can also use the odeintr package.

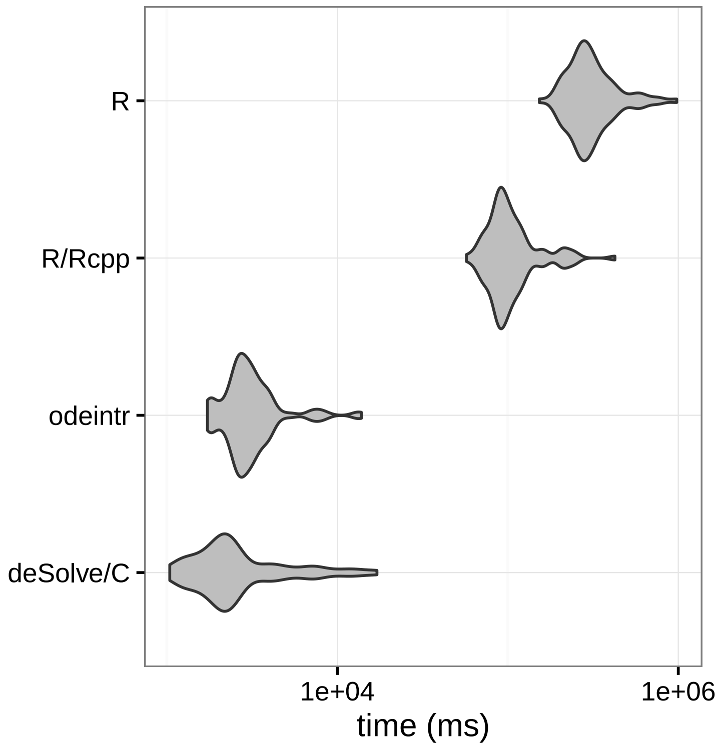
```
sir1_string = '
// variable N conflicts with a built-in ... so use NN
double foi = beta*x[0]/NN;
dxdt[0]=-foi*exp(x[1]);
dxdt[1]=foi-gamma;
'
## cat(JacobianCpp(sir))
## n.b. Jacobian doesn't work if we use intermediate foi variable
compile_sys("odeintr_sirgrad",sir1_string,pars=c("beta","gamma","NN"))
```

```
pars3 <- c(pars,i0=0.01)
tvec <- seq(0,20,by=0.01)
runODE.C <- function(t, params, method=c("C","Rcpp")) {
    method <- match.arg(method)
    odesol <- with(as.list(params),
                   ode(y=c(S=N,logI=log(N*i0)),
                       times=t,
                       func=switch(method,C="derivs",
                                   Rcpp=rcppgradfn),
                       parms=params[1:3],
                       dllname = switch(method,C="sirgrad",
                           Rcpp=file.path(s$buildDirectory,"sourceCpp_2.so")),
                       initfunc = switch(method,C="initmod",
                                         Rcpp=rcppinitfn),
                       nout = 1, outnames = character(0)))
    return(odesol[,"logI"])
}
runODE.R <- function(t, params, func=SIR.grad) {
    odesol <- with(as.list(params),
                   ode(times=t,
                       y=c(S=N,logI=log(N*i0)),
                       func=func,
                       parms=params[1:3]))
    return(odesol[,"logI"])
}
runODE.odeintr <- function(t, params) {
    ## isn't there a better way??
    do.call(environment(odeintr_sirgrad)$odeintr_sirgrad_set_params,
            unname(as.list(params[1:3])))
    ## unname to avoid having to do N -> NN switch
    res <- odeintr_sirgrad(init=with(as.list(params),c(N,log(N*i0))),
                    duration=diff(range(t)),
                    step_size=diff(t[1:2]),
                    start=t[1])
    return(res[,3])
}

stopifnot(all.equal(r1 <- runODE.R(tvec,pars3),runODE.C(tvec,pars3)))
r3 <- runODE.odeintr(tvec,pars3)
stopifnot(all.equal(r1,r3,tolerance=1e-6))
```

```
m2 <- microbenchmark(runODE.R(tvec,pars3),
                     runODE.R(tvec,pars3,func=sirgrad),
                     runODE.C(tvec,pars3),
                     runODE.odeintr(tvec,pars3))
levels(m2$expr) <- c("R",
                     "R/Rcpp",
                     "deSolve/C",
                     "odeintr")
m2$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 cost of R function calls (which in turn might call compiled C code), not the computation time for the gradient function itself:



- So far I haven't been able to figure out a way to compile Rcpp so that it can be used by `odesolve` directly, in the efficient way ... this won't matter so much if the gradient function is complex, or if we can use `odeintr` instead
- I haven't explored `odeintr` much. Looks like at present it only allows evenly spaced time points, but that may not be much of a limitation . . . also, it doesn't actually buy us much speed, at least in the simple SIR case

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.