Speeding up parameter inference with compiled models

Philipp H Boersch-Supan, Sadie J Ryan, and Leah R Johnson

September 2016

Differential equation (DE) models in R are easy to implement and allow simple interactive development using readable code and access to R's many high-level functions. This was part of our motivation to develop `deBInfer` as an R based inference package. However, numerically solving DE models specified as R functions is also relatively slow. The `deSolve` package also allows the evaluation of DE models that are defined in lower-level languages such as `C` and `FORTRAN`. These compiled models have the benefit of increased simulation speed. As the DE model is evaluated many times during the MCMC procedure, even moderate speed-ups from using compiled models can result in large absolute time savings.

We demonstrate the speed-up using an example from the `deSolve` documentation (Soetaert *et al.* 2010). Full details on the model specification can be found in Soetaert *et al.* (2009) which can be displayed with the command `vignette("compiledCode")`. Further details on the set up of the `deBInfer` inference procedure are described in Boersch-Supan *et al.* (2016) and annotated examples are available in the vignettes `vignette("logistic_ode_example")` and `vignette("vignette_chytrid_dede_example")`.

1 Specifying the ODE model

Following Soetaert *et al.* (2009), we use the following simple ODE:

$$\frac{dy_1}{dt} = -k_1 \cdot y_1 + k_2 \cdot y_2 \cdot y_3$$
$$\frac{dy_2}{dt} = k_1 \cdot y_1 - k_2 \cdot y_2 \cdot y_3 - k_3 \cdot y_2 \cdot y_2$$
$$\frac{dy_3}{dt} = k_3 \cdot y_2 \cdot y_2$$

where y_1, y_2 and y_3 are state variables, and k_1, k_2 and k_3 are parameters.

2 ODE model implementation in R

We implement this model as an `R` function:

```
modelR <- function(t, Y, parameters) {
  with (as.list(parameters),{

    dy1 <- -k1*Y[1] + k2*Y[2]*Y[3]
    dy3 <- k3*Y[2]*Y[2]
    dy2 <- -dy1 - dy3

    list(c(dy1, dy2, dy3))
  })
}
```

And also specify the Jacobian $\left(\frac{\partial y'}{\partial y}\right)$ for it:

```
jacR <- function (t, Y, parameters) {
  with (as.list(parameters),{

    PD[1,1] <- -k1
    PD[1,2] <- k2*Y[3]
    PD[1,3] <- k2*Y[2]
    PD[2,1] <- k1
    PD[2,3] <- -PD[1,3]
    PD[3,2] <- k3*Y[2]
    PD[2,2] <- -PD[1,2] - PD[3,2]

    return(PD)
  })
}
```
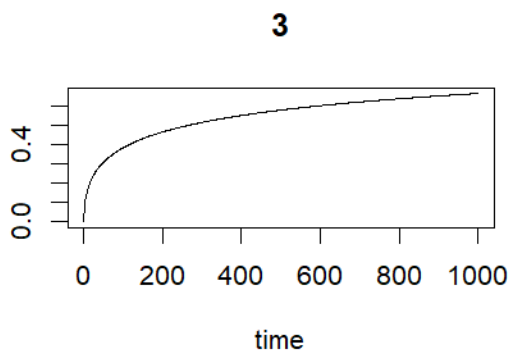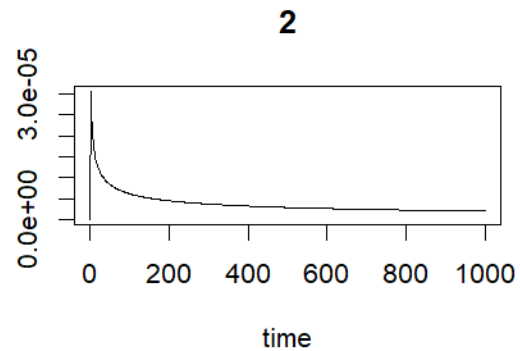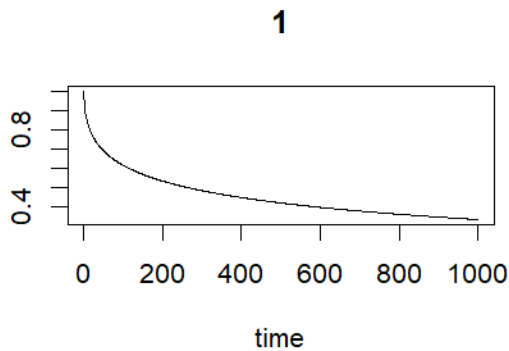
2.1 Solving R models

This model can be solved numerically in R using the deSolve package as follows.

```
library(deSolve)
parms <- c(k1 = 0.04, k2 = 1e4, k3=3e7)
Y     <- c(1.0, 0.0, 0.0)
times <- seq(0, 0.1*10^4, length.out = 1000)
PD    <- matrix(nrow = 3, ncol = 3, data = 0)
outR   <- ode(Y, times, modelR, parms = parms, jacfunc = jacR)
plot(outR)
```

3   ODE model implementation in C

To create compiled models (dynamic-link libraries (.dll) on Windows or shared objects (.so) on other systems) a recent version of the GNU compiler suite is required. GNU compilers are included by default in most Linux distributions. Windows users can install the required toolchain by following the instructions on [https://cran.r-project.org/bin/windows/Rtools/]. OSX users need to download and install an appropriate version of Xcode from the Apple developer website [https://developer.apple.com/] or the OSX App Store.

The call to the derivative and Jacobian function is more complex for compiled code compared to R-code, because it has to comply with the interface needed by the integrator. The requirements for this are detailed in Soetaert *et al.* (2009). A C implementation of the example model is found below.

```c
/* file mymod.c */
#include <R.h>
static double parms[3];
#define k1 parms[0]
#define k2 parms[1]
#define k3 parms[2]


/* initializer  */
void initmod(void (* odeparms)(int *, double *))
{
    int N=3;
    odeparms(&N, parms);
}
```

```c
/* Derivatives and 1 output variable */
void derivs (int *neq, double *t, double *y, double *ydot,
             double *yout, int *ip)
{

    if (ip[0] <1) error("nout should be at least 1");
    ydot[0] = -k1*y[0] + k2*y[1]*y[2];
    ydot[2] = k3 * y[1]*y[1];
    ydot[1] = -ydot[0]-ydot[2];

    yout[0] = y[0]+y[1]+y[2];
}

/* The Jacobian matrix */
void jac(int *neq, double *t, double *y, int *ml, int *mu,
         double *pd, int *nrowpd, double *yout, int *ip)
{
  pd[0]               = -k1;
  pd[1]               = k1;
  pd[2]               = 0.0;
  pd[(*nrowpd)]       = k2*y[2];
  pd[(*nrowpd) + 1]   = -k2*y[2] - 2*k3*y[1];
  pd[(*nrowpd) + 2]   = 2*k3*y[1];
  pd[(*nrowpd)*2]     = k2*y[1];
  pd[2*(*nrowpd) + 1] = -k2 * y[1];
  pd[2*(*nrowpd) + 2] = 0.0;
}
/* END file mymod.c */
```

3.1 Compiling and solving ODE models implemented in C

To run the C implementation of the model it must first be compiled. This can be done using the R system statement

```r
system("R CMD SHLIB mymod.c")
```

which will create the file mymod.dll on Windows, or mymod.so on other platforms.

We can then load the compiled model with

```r
dyn.load(paste("mymod", .Platform$dynlib.ext, sep = ""))
```
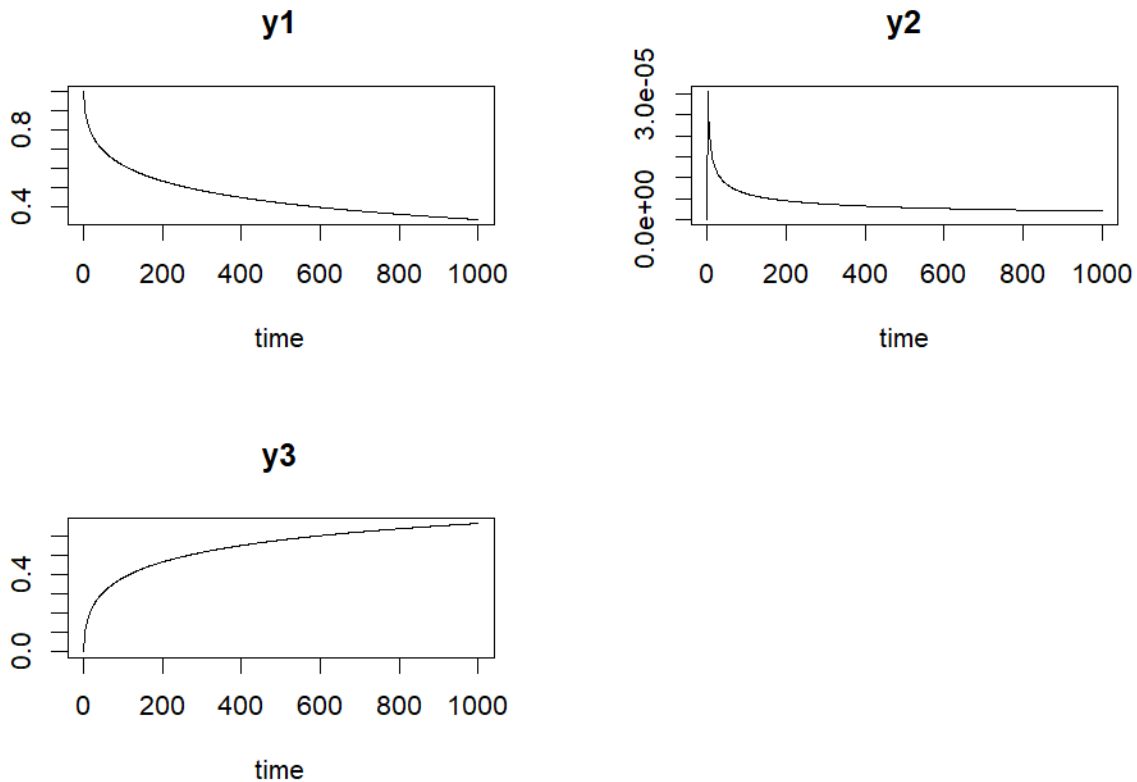
The compiled model can be solved numerically from R using the deSolve package as follows.

```r
parms <- c(k1 = 0.04, k2 = 1e4, k3=3e7)
Y     <- c(y1 = 1.0, y2 = 0.0, y3 = 0.0)
times <- seq(0, 0.1*10^4, length.out = 1000)

out <- ode(Y, times, func = "derivs", parms = parms,
           jacfunc = "jac", dllname = "mymod",
           initfunc = "initmod", nout = 1, outnames = "Sum")
plot(out, which = 1:3)
```

y1

y2

y3

4 Comparing solver speed

We can determine the speed improvement as a function of desired output time points using the `microbenchmark` package

```
#Set up a vector of the number of solver time points
timepoints <- c(10,100, 500, 1000,2000, 5000, 10000)
#solve both model version on the desired time points
solve_time <- lapply(timepoints, function(x){
                times <- seq(0, 0.1*10^4, length.out = x)
                comp <- microbenchmark::microbenchmark(
                  #solve the C model
                  C = ode(Y, times, func = "derivs", parms = parms,
                          jacfunc = "jac", dllname = "mymod",
                          initfunc = "initmod", nout = 1, outnames = "Sum"),
                  #solve the R model
                  R = ode(Y, times, modelR, parms = parms, jacfunc = jacR), times=10
                  )
                return(comp)
                }
              )

#save simulation if desired
#saveRDS(solve_time, "examples/Soetaert-r-vs-c-solve_time.RDS")
```

```r
#calculate summaries from the simulations
solve_speedup <- sapply(solve_time,
                        function(x)mean(x$time[x$expr=="R"]/x$time[x$expr=="C"]))
solve_R <- sapply(solve_time, function(x)mean(x$time[x$expr=="R"]/1e9))
solve_C <- sapply(solve_time, function(x)mean(x$time[x$expr=="C"]/1e9))

#plot the results
par(mfrow=c(1,2))
plot(timepoints, solve_R, type="p", ylab = "Execution time (s)", xlab = "Time points")
points(timepoints, solve_C, col="red", type="p", pch=c(16))
legend("topleft", legend = c("R ODE", "C ODE"), pch = c(1,16), col= c("black", "red"))
plot(rep(timepoints, each=1), solve_speedup, type="p", ylab = "Relative execution time"
     , xlab = "Time points", ylim = c(0,max(solve_speedup)))
```
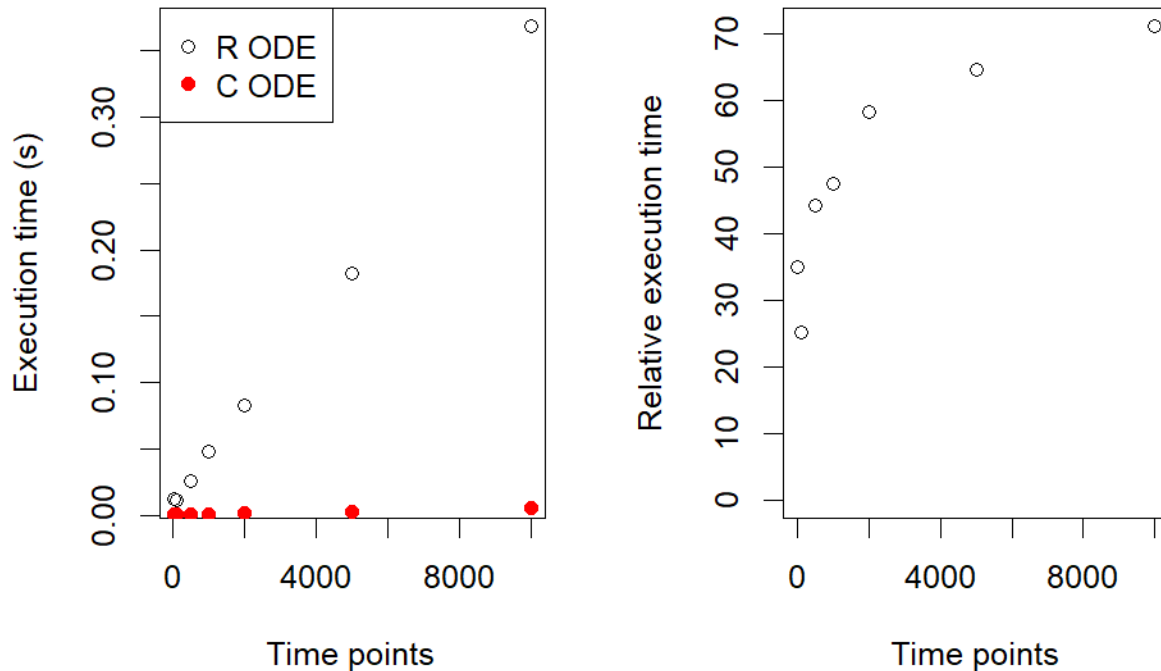

which shows us that the `C` version of the models is solved 25-71 times faster than the `R` version.

5 Conducting inference with compiled models

We can now simulate a dataset to demonstrate the inference procedure:

```r
set.seed(143)
#sample from integration output. Force include the first 100 time-points
#to aid parameter identifiability
obs <- as.data.frame(outR[c(1:100,runif(100, 0, nrow(outR))),])
#order the observations
```

```
obs <- obs[order(obs$time),]
#plot the observations
par(mfrow=c(1,3))
plot(obs$time, obs[,2], type='p', ylab="y1", xlab="Time")
plot(obs$time, obs[,3], type='p', ylab="y2", xlab="Time")
plot(obs$time, obs[,4], type='p', ylab="y3", xlab="Time")
```
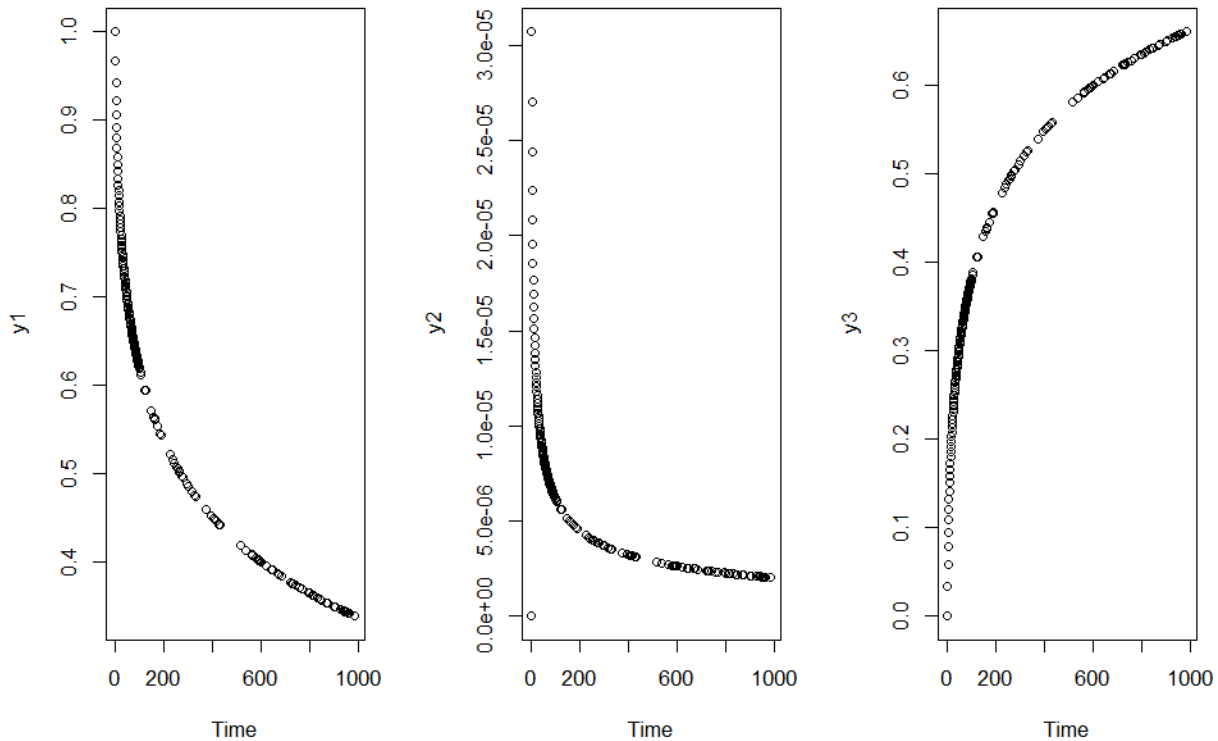

5.1 Defining an observation model and parameters for inference

For simplicity we assume a normal log-likelihood for these data

$$\ell(\mathcal{Y}|\boldsymbol{\theta}) = \sum_i \sum_t \ln\left(\frac{1}{\sigma_{obs}\sqrt{2\pi}} \exp\left(-\frac{(\tilde{y}_{t,i} - (y_{t,i})^2}{2\sigma_{obs}^2}\right)\right) \tag{1}$$

where $\tilde{y}_{t,i}$ are the observations, and $y_{t,i}$ are the predictions of the DE model given the current MCMC sample of the parameters $\boldsymbol{\theta}$.

```
# the observation model
obs_model <- function(data, sim.data, samp){

  llik.y1 <- sum(dnorm(obs[,2], mean = sim.data[,2], sd = samp[['sd.y1']], log = TRUE))
  llik.y2 <- sum(dnorm(obs[,3], mean = sim.data[,3], sd = samp[['sd.y2']], log = TRUE))
  llik.y3 <- sum(dnorm(obs[,4], mean = sim.data[,4], sd = samp[['sd.y3']], log = TRUE))
  return(llik.y1 + llik.y2 + llik.y3)
}
```

7

We declare the DE model parameters and their associated priors and MCMC parameters. Note that we are using the asymmetric uniform proposal distribution $\mathcal{U}(\frac{a}{b}\theta^{(k)}, \frac{b}{a}\theta^{(k)})$ for the variance parameters (`samp.type="rw-unif"`), as this ensures strictly positive proposals.

```r
library(deBInfer)
k1 <- debinfer_par(name = "k1", var.type = "de", fixed = FALSE,
                value = 0.05, prior = "lnorm", hypers = list(meanlog = 0, sdlog = 1),
                 prop.var = c(998,1000), samp.type = "rw-unif")


k2 <- debinfer_par(name = "k2", var.type = "de", fixed = FALSE,
                value = 8000, prior = "norm", hypers = list(mean = 5000, sd = 1000),
                prop.var = 500, samp.type="rw")
k3 <- debinfer_par(name = "k3", var.type = "de", fixed = FALSE,
                value = 3e7, prior = "norm", hypers = list(mean = 1e7, sd = 5e6),
                prop.var = 5e7, samp.type="rw")


sd.y1 <- debinfer_par(name = "sd.y1", var.type = "obs", fixed = FALSE,
                value = 0.05, prior = "lnorm", hypers = list(meanlog = 0, sdlog = 1),
                prop.var = c(3,4), samp.type = "rw-unif")
sd.y2 <- debinfer_par(name = "sd.y2", var.type = "obs", fixed = FALSE,
                value = 0.05, prior = "lnorm", hypers = list(meanlog = 0, sdlog = 1),
                prop.var = c(3,4), samp.type = "rw-unif")
sd.y3 <- debinfer_par(name = "sd.y3", var.type = "obs", fixed = FALSE,
                value = 0.05, prior = "lnorm", hypers = list(meanlog = 0, sdlog = 1),
                prop.var = c(3,4), samp.type = "rw-unif")
```

Lastly, we provide initial values for the DE:

```r
y1 <- debinfer_par(name = "y1", var.type = "init", fixed = TRUE, value = 1)
y2 <- debinfer_par(name = "y2", var.type = "init", fixed = TRUE, value = 0)
y3 <- debinfer_par(name = "y3", var.type = "init", fixed = TRUE, value = 0)
```

The declared parameters are then collated using the `setup_debinfer` function. Note that for models with more than one state variable, **the initial values must be entered in the same order, as they are specified in the DE model function**, as the solver matches these values by position, rather than by name. More details can be found in `?deSolve::ode`. The remaining parameters can be entered in any order for the R based inference, **but are matched by position in the compiled model**.

```r
mcmc.pars <- setup_debinfer(k1, k2, k3, sd.y1, sd.y2, sd.y3, y1, y2, y3)
```

5.2 Conduct inference

Finally we use deBInfer to estimate the parameters of the original model using both the R and the C version of the model.

```r
# do inference with deBInfer
  # MCMC iterations
iter <- 1000
```

For the R model, the inference call proceeds as outlined in Boersch-Supan *et al.* (2016), although we wrap it in a `system.time` call to measure the execution time.

```r
  # inference call
Rt <- system.time(
  mcmc_samples <- de_mcmc(N = iter, data = obs, de.model = modelR,
                        obs.model = obs_model, all.params = mcmc.pars,
```

```
                        Tmax = max(obs$time), data.times = obs$time, cnt = 500,
                        plot = FALSE, solver = "ode", verbose.mcmc = FALSE)
  )
```

```
## Order of initial conditions is  y1, y2, y3
```

```
## initial posterior probability = 568.058099497889
```

```
print(Rt)
```

```
##    user  system elapsed
##   39.33    0.00   39.41
```

For the C model we pass in the name of the C function that calculates the derivatives as the `de.model` parameter, and furthermore provide the additional parameters `initfunc`, `jacfunc`, `dllname`, `nout`, and `outnames` to the solver. Again we wrap the inference call in `system.time` to measure execution time.

```
Ct <- system.time(
  mcmc_samplesC <- de_mcmc(N = iter, data = obs, de.model = "derivs",
                        jacfunc = "jac", dllname = "mymod",
                        initfunc = "initmod", nout = 1, outnames = "Sum",
                        obs.model = obs_model, all.params = mcmc.pars,
                        Tmax = max(obs$time), data.times = obs$time, cnt = 1250,
                        plot = FALSE, solver = "ode", verbose.mcmc = FALSE)
  )
```

```
## Order of initial conditions is  y1, y2, y3
```

```
## initial posterior probability = 568.058099497889
```

```
print(Ct)
```

```
##    user  system elapsed
##    4.87    0.00    4.89
```

We can see that inference using the C model is approximately 8 times faster than inference using the R model.

6   Speed up of the inference procedure

We can see that inference and posterior simulations using the compiled model are substantially faster. This speed up is consistent across a range of MCMC iterations:

```
iter_reps <- rep(c(100,1000,2000,5000,10000),3)
timings <- matrix(nrow=length(iter_reps),ncol=2)

for (i in seq_along(iter_reps)){

  timings[i,1] <- system.time(mcmc_samples <- de_mcmc(N = iter_reps[i], data = obs,
                        de.model = modelR,
                        obs.model = obs_model, all.params = mcmc.pars,
                        Tmax = max(obs$time), data.times = obs$time, cnt = 500,
                        plot = FALSE, solver = "ode", verbose.mcmc = FALSE))[3]
  timings[i,2] <- system.time(mcmc_samplesC <- de_mcmc(N = iter_reps[i], data = obs,
                        de.model = "derivs",
                        jacfunc = "jac", dllname = "mymod",
                        initfunc = "initmod", nout = 1, outnames = "Sum",
                        obs.model = obs_model, all.params = mcmc.pars,
```

```
                              Tmax = max(obs$time), data.times = obs$time, cnt = 1250,
                              plot = FALSE, solver = "ode", verbose.mcmc = FALSE))[3]
}
#saveRDS(timings, "examples/Soetaert-r-vs-c-timings.RDS")
```

```
iter_reps <- rep(c(100,1000,2000,5000,10000),3)
if(!exists("timings")) timings <- readRDS("Soetaert-r-vs-c-timings.RDS")
#pdf("examples/c-vs-r-iters.pdf")
par(mfrow=c(1,2))
plot(iter_reps, timings[,1], ylim=c(0, max(timings[,1])), type="p",
     ylab = "Execution time (s)", xlab = "MCMC iterations")
points(iter_reps, timings[,2], col="red", type="p", pch=c(1,16))
legend("topleft", legend = c("R ODE", "C ODE"), pch = c(1,16), col= c("black", "red"))
plot(iter_reps, timings[,1]/timings[,2], ylim=c(0, max(timings[,1]/timings[,2])), type="p",
     ylab = "Relative execution time", xlab = "MCMC iterations")
```
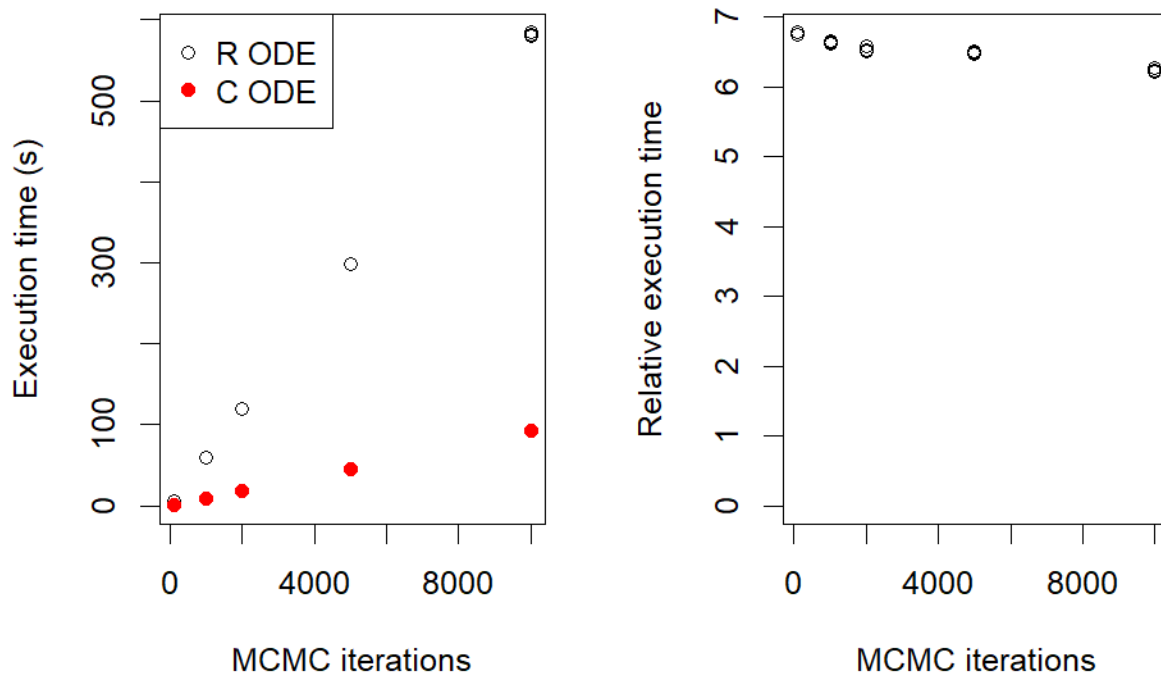


```
#dev.off()
```

7   Speed up of posterior simulations

The use of compiled models also speeds up the simulation of the posterior trajectories. The call to `post_sim` for the compiled model requires the same additional arguments as outlined above for `de_mcmc`.

```
#Simulate using the R model and the samples from the inference on it
Rs <- system.time(
```

```
  post_traj <- post_sim(mcmc_samples, n=100, times=0:1000, burnin=100,
                                    output = 'all', prob = 0.95)
  )
print(Rs)

##     user  system elapsed
##     4.87    0.01    4.91
```

```
#Simulate using the C model and the samples from the inference on it
Cs <- system.time(
  post_trajC <- post_sim(mcmc_samplesC, n=100, times=0:1000, burnin=100,
                                    output = 'all', prob = 0.95, jacfunc = "jac",
                                    dllname = "mymod",initfunc = "initmod", nout = 1,
                                    outnames = "Sum")
  )
print(Cs)

##     user  system elapsed
##     0.60    0.05    0.64
```

We can see that simulations using the C model are approximately 8 times faster than simulations using the R model.

Finally we can plot the posterior median trajectories for both inference runs to visually confirm that both the C and R model are estimated correctly.

```
#Plot simulations and observations
par(mfrow=c(1,3))
plot(post_traj$time, post_traj$median$y1, type='l',lwd=1, ylab = "y1", xlab = "Time")
lines(post_trajC$time, post_trajC$median$y1, col="red", lty=3, lwd=1)
points(obs$time, obs[,2])
legend("topright", legend = c("Posterior R", "Posterior C", "Observations"),
       lty=c(1,3,NA), pch=c(NA,NA,1), col=c("black","red","black"))
plot(post_traj$time, post_traj$median$y2, type='l', ylab = "y1", xlab = "Time")
lines(post_trajC$time, post_trajC$median$y2, col="red", lty=3, lwd=1)
points(obs$time, obs[,3])
plot(post_traj$time, post_traj$median$y3, type='l', ylab = "y1", xlab = "Time")
lines(post_trajC$time, post_trajC$median$y3, col="red", lty=3, lwd=1)
points(obs$time, obs[,4])
```
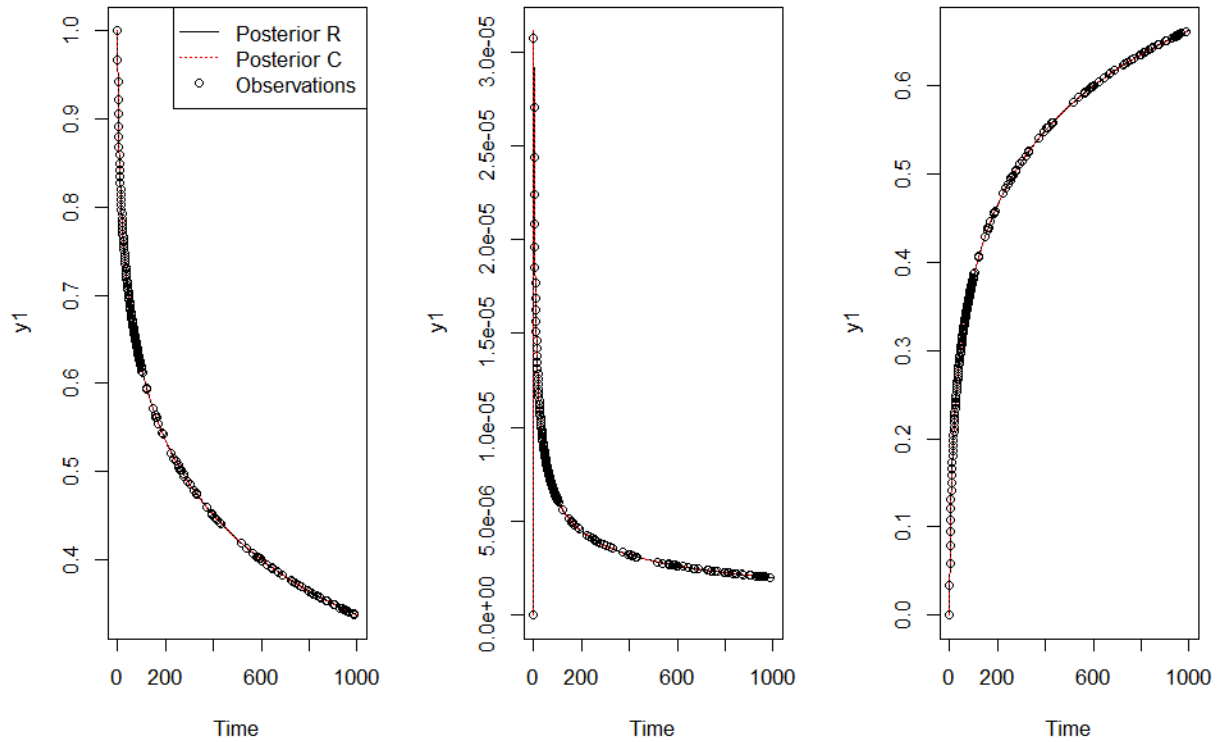
References

Boersch-Supan, P.H., Ryan, S.J. & Johnson, L.R. (2016). deBInfer: Bayesian inference for dynamical models of biological systems. *Methods in Ecology and Evolution*, **DOI: 10.1111/2041-210X.12679**.

Soetaert, K., Petzoldt, T. & Setzer, R.W. (2009). *R package desolve, writing code in compiled languages.*

Soetaert, K., Petzoldt, T. & Setzer, R.W. (2010). Solving differential equations in R: Package deSolve. *Journal of Statistical Software*, **33**, 1–25.