Speeding up parameter inference with compiled models

Philipp H Boersch-Supan and Leah R Johnson

September 12, 2016

DE models in R are easy to implement, allow simple interactive development using readable code and access to Rs high-level procedures. All of which were part of our motivation to develop `deBInfer` as an R based inference package. However, 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 may result in large absolute time savings.

We demonstrate the speed-up for an example from the `deSolve` documentation (Soetaert *et al.* 2010,Soetaert et al. (2009))

1 A simple ODE example

Assume the following simple ODE (which is from the `LSODA` source code):

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

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

We first implement and run this model in pure `R`, then show how to do this in `C`.

1.1 ODE model implementation in R

An ODE model implemented in pure `R` should be defined as:

```
yprime = func(t, y, parms, ...)
```

where `t` is the current time point in the integration, `y` is the current estimate of the variables in the ODE system, and `parms` is a vector or list containing the parameter values. The optional dots argument (`...`) can be used to pass any other arguments to the function. The return value of `func` should be a list, whose first element is a vector containing the derivatives of `y` with respect to time, and whose next elements contain output variables that are required at each point in time.

The `R` implementation of the simple ODE is given below:

```
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))
  })
}
```

The Jacobian ($\frac{\partial y'}{\partial y}$) associated to the above example is:

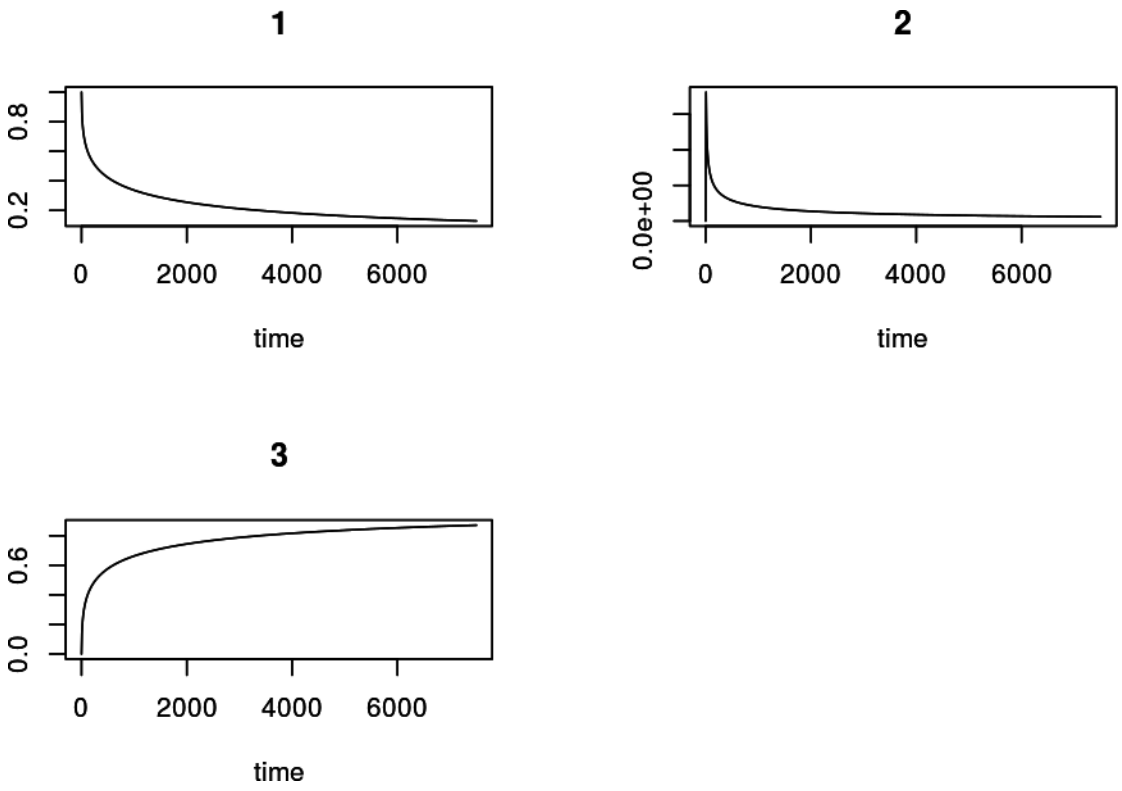
```
jacR <- function (t, Y, parameters) {
  with (as.list(parameters),{

    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)
  })
}
```

This model can then be run as follows:

```
library(deSolve)
parms <- c(k1 = 0.04, k2 = 1e4, k3=3e7)
Y     <- c(1.0, 0.0, 0.0)
times <- seq(0, 0.75*10^4, length.out = 1000)
PD    <- matrix(nrow = 3, ncol = 3, data = 0)
outR   <- ode(Y, times, modelR, parms = parms, jacfunc = jacR)
plot(outR)
```



1.2 ODE model implementation in C

In order to create compiled models (.DLL = dynamic link libraries on Windows or .so = shared objects on other systems) you must have a recent version of the GNU compiler suite installed, which is quite standard for Linux. Windows users find all the required tools on [http://www.murdoch-sutherland.com/Rtools/]. Getting DLLs produced by other compilers to communicate with R is much more complicated and therefore not recommended. More details can be found on [http://cran.r-project.org/doc/manuals/R-admin.html].

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 source codes.

Below is an implementation of this model in `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);
}

/* 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 */
```

The structure of the implementation in `C` is described in detail in Soetaert et al. (2009).

2 Running ODE models implemented in compiled code

To run the model described above, the `C` code must first be compiled[1]. This can be done using the `R system` statement

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

This will create file `mymod.dll` on Windows, or `mymod.so` on other platforms.

We load the compiled model with

```
dyn.load(paste("mymod", .Platform$dynlib.ext, sep = ""))
```

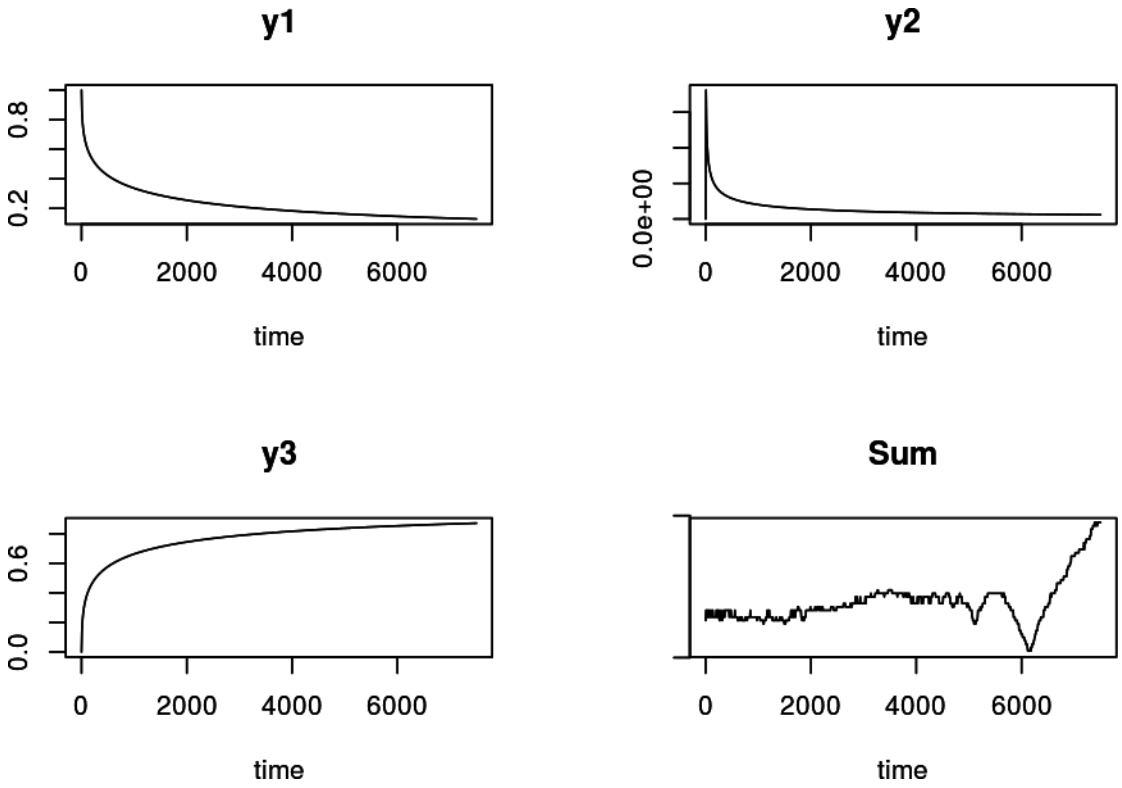
The model can now be run as follows:

```
parms <- c(k1 = 0.04, k2 = 1e4, k3=3e7)
Y     <- c(y1 = 1.0, y2 = 0.0, y3 = 0.0)
times <- seq(0, 0.75*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)
```

[1] This requires a correctly installed GNU compiler.

```
## Warning in plot.window(...): relative range of values = 31 * EPS, is small
## (axis 2)
```



We can determine the speed improvement using the `microbenchmark` package, which shows us that the `C` version of the models is solved about 10 times faster than the `R` version.

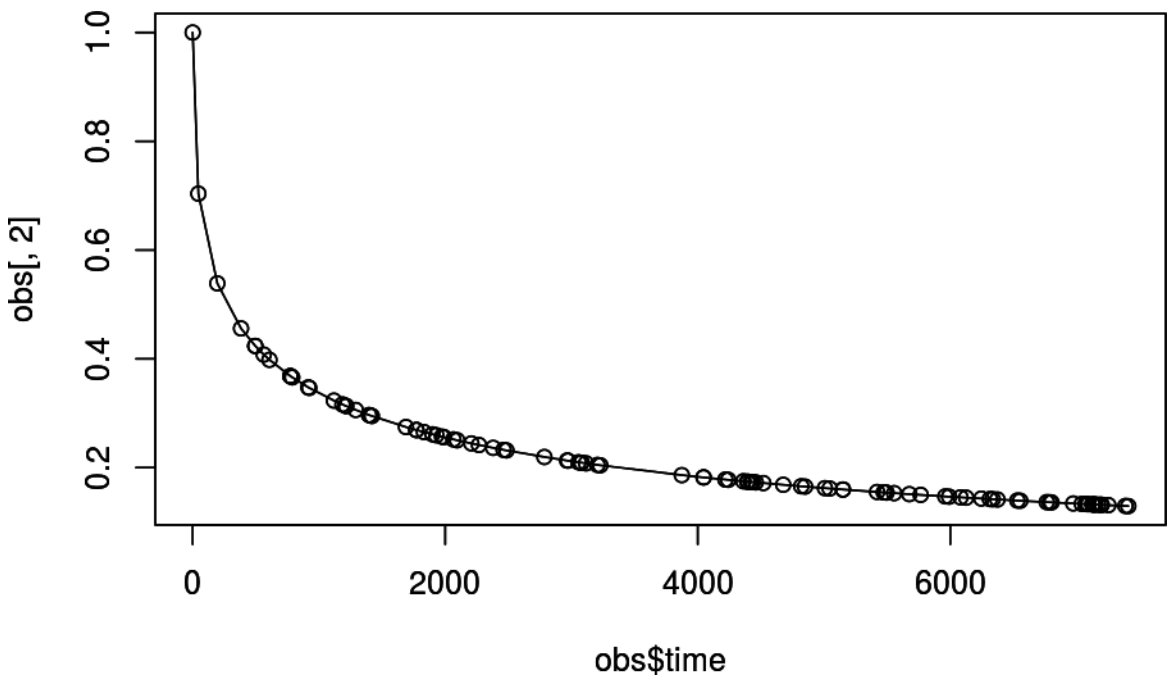
```
comp <- microbenchmark::microbenchmark(
  C = ode(Y, times, func = "derivs", parms = parms,
          jacfunc = "jac", dllname = "mymod",
          initfunc = "initmod", nout = 1, outnames = "Sum"),
  R = ode(Y, times, modelR, parms = parms, jacfunc = jacR)
)
print(comp, unit = "ms")
```

```
## Unit: milliseconds
##  expr        min        lq       mean     median         uq        max
##     C   3.099627   3.13102   3.251667   3.266373   3.308068   5.318275
##     R 109.454282 111.85825 112.454730 112.015886 112.186439 137.195240
##  neval
##    100
##    100
```

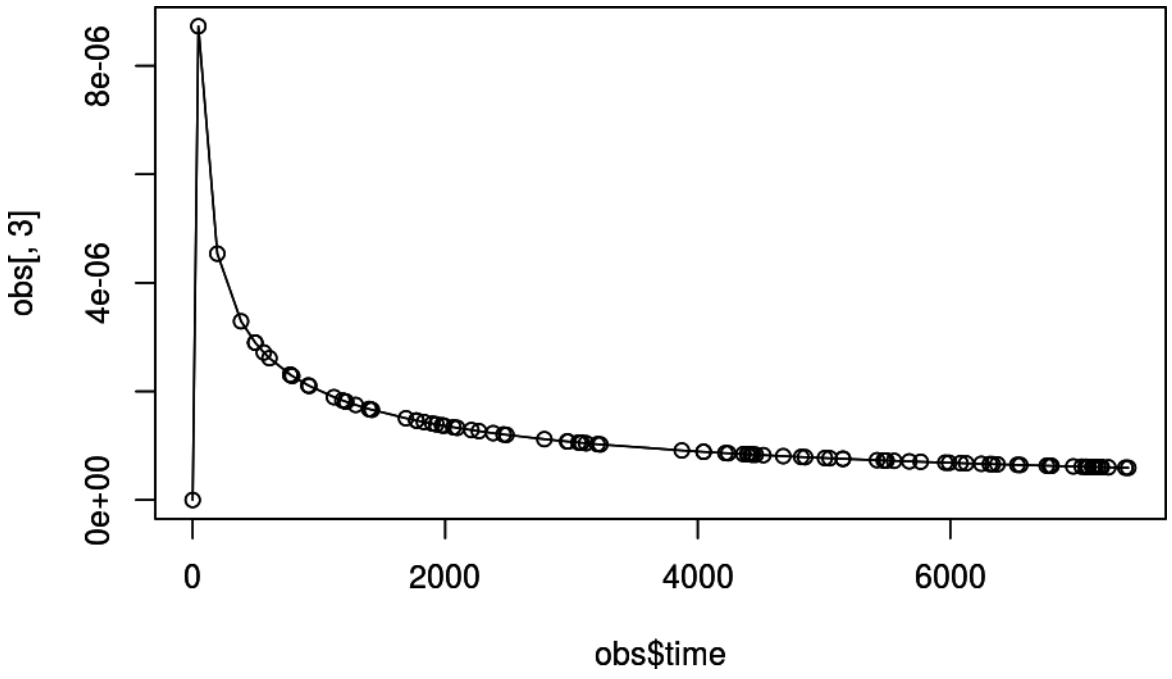
The Again, please refer to the `deSolve` documentation (Soetaert *et al.* 2009) for a detailed description of the syntax.

We can now simulate a dataset to demonstrate the inference procedure

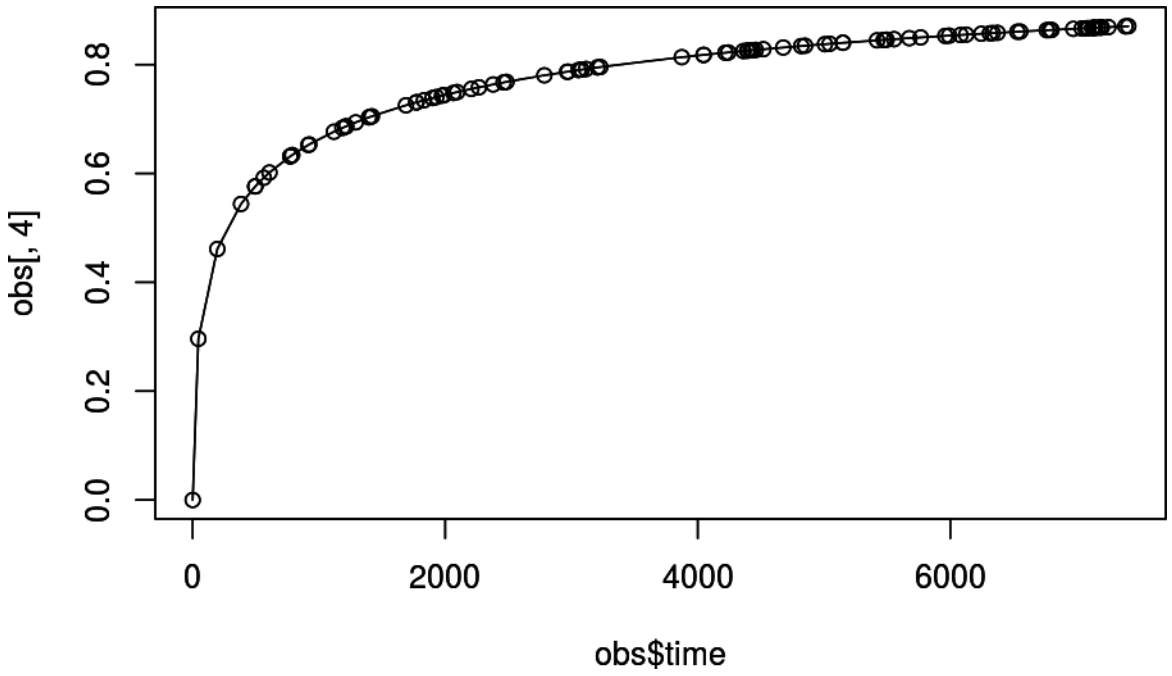
```r
set.seed(143)
#force include the first time-point (t=0)
obs <- as.data.frame(outR[c(1,runif(100, 0, nrow(outR))),])
obs <- obs[order(obs$time),]
plot(obs$time, obs[,2], type='o')
```




```r
plot(obs$time, obs[,3], type='o')
```



```
plot(obs$time, obs[,4], type='o')
```



3 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)
}
```

We declare the DE model parameter r, assign a prior $r \sim \mathcal{N}(0,1)$ and a random walk sampler with a Normal kernel (`samp.type="rw"`) and proposal variance of 0.005. Similarly, we declare $K \sim \ln\mathcal{N}(1,1)$ and $\ln(\sigma_{obs}^2) \sim \mathcal{N}(0,1)$. 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 parameter (`samp.type="rw-unif"`), as this ensures strictly positive proposals.

```
library(deBInfer)
k1 <- debinfer_par(name = "k1", var.type = "de", fixed = FALSE,
                   value = 0.1, prior = "norm", hypers = list(mean = 0, sd = 1),
```

```
                prop.var = 0.00001, samp.type="rw")

k2 <- debinfer_par(name = "k2", var.type = "de", fixed = FALSE,
                value = 5000, prior = "norm", hypers = list(mean = 5000, sd = 500),
                prop.var = 10, samp.type="rw")
k3 <- debinfer_par(name = "k3", var.type = "de", fixed = FALSE,
                value = 1e7, prior = "norm", hypers = list(mean = 1e7, sd = 1e6),
                prop.var = 5e6, 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 an initial value $N_0 = 0.1$ for the DE:

```
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.

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

4 Conduct inference

Finally we use deBInfer to estimate the parameters of the original model. `de_mcmc` is the workhorse of the package and runs the MCMC estimation. The progress of the MCMC procedure can be monitored using the `cnt`, `plot` and `verbose` options: Every `cnt` iterations the function will print out information about the current state, and, if `plot=TRUE`, traceplots of the chains will be plotted. Setting `verbose=TRUE` will print additional information. Note that frequent plotting will substantially slow down the MCMC sampler, and should be used only on short runs when tuning the sampler.

```
# do inference with deBInfer
  # MCMC iterations
iter <- 1000
  # 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 = -372.440801942676
```

```
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 = 50,
                          plot = FALSE, solver = "ode", verbose.mcmc = FALSE))
```

```
## Order of initial conditions is  y1, y2, y3
## initial posterior probability = -372.440801942676
```

```
print(Rt)
```

```
##    user  system elapsed
## 118.907   0.032 119.021
```

```
print(Ct)
```

```
##    user  system elapsed
##  13.192   0.008  13.209
```

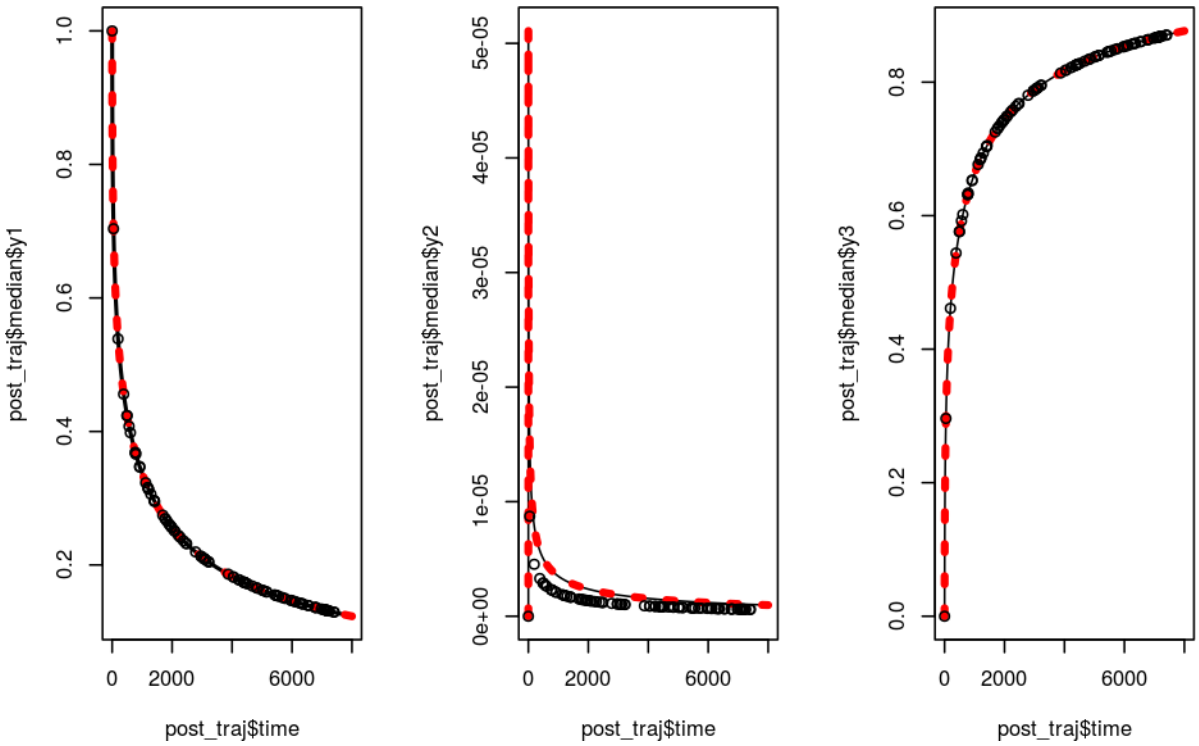
```
system.time(post_traj <- post_sim(mcmc_samples, n=100, times=0:8000, burnin=100, output = 'all', prob =
```

```
##    user  system elapsed
##  81.222   0.076  81.332
```

```
system.time(post_trajC <- post_sim(mcmc_samplesC, n=100, times=0:8000, burnin=100, output = 'all', prob
```

```
##    user  system elapsed
##  11.893   0.068  11.966
```

```
par(mfrow=c(1,3))
plot(post_traj$time, post_traj$median$y1, type='l',lwd=2)
lines(post_trajC$time, post_trajC$median$y1, col="red", lty=3, lwd=4)
points(obs$time, obs[,2])
plot(post_traj$time, post_traj$median$y2, type='l')
lines(post_trajC$time, post_trajC$median$y2, col="red", lty=3, lwd=4)
points(obs$time, obs[,3])
plot(post_traj$time, post_traj$median$y3, type='l')
lines(post_trajC$time, post_trajC$median$y3, col="red", lty=3, lwd=4)
points(obs$time, obs[,4])
```



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.