ODEs and Nimble

Motivation

Here we show how to fit ODE-type models to noisy data using `Nimble`. We begin with writing a `C++` function encoding the ODE system and allowing for its numerical resolution. This step is done using the package `odeintr`. Then we define a `Nimble` function to wrap the `C++` ODE solver. We start with a 1D ODE example, then move to a 4D ODE for fun. Last, we fit a model that is specified with a system of ODEs and gaussian noise.

Let's load the packages:

```
library(nimble)
library(odeintr)
library(Rcpp)
source("odetoNimble_fun.R")
```

1D example

Our first example is a simple logistic growth model $dy/dt = ry(1 - y/K)$. We define the system of equation then we build the `C++` code and compile it:

```
logistic.sys = "dxdt[0] = alpha * x[0] * (1-x[0]/beta)"
myModel <- odetoNimble("logistic", logistic.sys, pars=c("alpha","beta"))
```

Now we use the `C++`model to define a new `Nimble`function:

```
Cmodel_logistic <- compileNimble(myModel, showCompilerOutput = TRUE)
```

```
## clang++ -mmacosx-version-min=10.13 -std=gnu++11 -I"/Library/Frameworks/R.framework/Resources/include
## clang++ -mmacosx-version-min=10.13 -std=gnu++11 -dynamiclib -Wl,-headerpad_max_install_names -undefi
## warning: unknown warning option '-Wno-misleading-indentation'; did you mean '-Wno-binding-in-conditi
## 1 warning generated.
## clang++ -mmacosx-version-min=10.13 -std=gnu++11 -I"/Library/Frameworks/R.framework/Resources/include
## clang++ -mmacosx-version-min=10.13 -std=gnu++11 -dynamiclib -Wl,-headerpad_max_install_names -undefi
## warning: unknown warning option '-Wno-misleading-indentation'; did you mean '-Wno-binding-in-conditi
## 1 warning generated.
```

Set some values:

```
y <- 0.1 # initial value
times <- seq(from = 0, to = 10, by = 0.01) # time sequence
params <- c(1.5, 10) # r and K
time_step = 0.01
```

Lets run the new function:

```
resultat <- Cmodel_logistic(y, times, time_step,par=params)
head(resultat)
```

```
##      [,1]      [,2]
## [1,] 0.00 0.1000000
## [2,] 0.01 0.1014960
## [3,] 0.02 0.1030141
## [4,] 0.03 0.1045547
## [5,] 0.04 0.1061180
## [6,] 0.05 0.1077045
```

We now define a `Nimble` code:

```
demoCode <- nimbleCode({
  init ~ dunif(0.1,0.1)
  y[1:1001,1:2] <- myModel(c(init),times[1:1001],0.01,param[1:2])
})
```

Build and compile le model:

```
constants = list()
data = list()
inits = list(init=0.1,param=c(alpha=1.5,beta=10),times=times)
demoModel <- nimbleModel(demoCode, constants, data, inits, check = TRUE, calculate = FALSE)
```

```
## Error in nimbleConvert(initR) :
##   impossible de trouver la fonction "nimbleConvert"
## Error in nimbleConvert(initR) :
##   impossible de trouver la fonction "nimbleConvert"
## Note: cannot calculate logProb for node  y[1:1001, 1:2] .
## NAs were detected in model variable: y.
## NaNs were detected in model variable: logProb_init.
```

```
CdemoModel <- compileNimble(demoModel, dirName=file.path(getwd(),"test"), showCompilerOutput = TRUE)
```

```
## clang++ -mmacosx-version-min=10.13 -std=gnu++11 -I"/Library/Frameworks/R.framework/Resources/include
## clang++ -mmacosx-version-min=10.13 -std=gnu++11 -I"/Library/Frameworks/R.framework/Resources/include
## clang++ -mmacosx-version-min=10.13 -std=gnu++11 -dynamiclib -Wl,-headerpad_max_install_names -undefi
## warning: unknown warning option '-Wno-misleading-indentation'; did you mean '-Wno-binding-in-conditi
## 1 warning generated.
## warning: unknown warning option '-Wno-misleading-indentation'; did you mean '-Wno-binding-in-conditi
## 1 warning generated.
```

```
CdemoModel$calculate()
```

```
## [1] NaN
```

```
head(CdemoModel$y)
```

```
##      [,1]      [,2]
## [1,] 0.00 0.1000000
## [2,] 0.01 0.1014960
## [3,] 0.02 0.1030141
## [4,] 0.03 0.1045547
## [5,] 0.04 0.1061180
## [6,] 0.05 0.1077045
```

4D example

I use data from a paper by Witkowski and Brais entitled Bayesian Analysis of Epidemics - Zombies, Influenza, and other Diseases. The authors provide a Python notebook here. Briefly speaking, they counted the number of living deads in several famous zombie movies. I use the data from Shaun of the Dead.

The authors propose the following SIR epidemic model (more here) to capture the zombie apocalypse:

$$dS/dt = -\beta SZ dE/dt = \beta SZ - \zeta E dZ/dt = \zeta E - \alpha SZ dR/dt = \alpha SZ$$

Let us solve this system of ODEs. First, set up some values:

```
y <- c(508.2, 0, 1, 0) # intial values
times <- seq(from = 0, to = 50, by = 1) # time sequence
params <- c(0.2, 6, 0, 508.2) # beta, zeta, alpha, Sinit
```

Write the system of ODEs:

```
zombie.sys = '
  dxdt[0] = -beta * x[0] * x[2] / Sinit;
  dxdt[1] = beta * x[0] * x[2] / Sinit - zeta * x[1];
  dxdt[2] = zeta * x[1] - alpha * x[0] * x[2];
  dxdt[3] = alpha * x[0] * x[2];
'
myModel <- odetoNimble("zombie", zombie.sys, pars=c("beta","zeta","alpha","Sinit"))
```

Write the `Nimble` code:

```
code <- nimbleCode({
  xOde[1:51, 1:5] <- myModel(y[1:4], times[1:51],1, params[1:4])
})
constants <- list()
data <- list()
inits <- list(y = y, params = params, times=times)
demoModel <- nimbleModel(code, constants, data, inits, check = TRUE, calculate = FALSE)
```

```
## Error in nimbleConvert(initR) :
##   impossible de trouver la fonction "nimbleConvert"
## Error in nimbleConvert(initR) :
##   impossible de trouver la fonction "nimbleConvert"
## Note: cannot calculate logProb for node  xOde[1:51, 1:5] .
## NAs were detected in model variable: xOde.
```

```
CdemoModel <- compileNimble(demoModel, dirName=file.path(getwd(),"test"), showCompilerOutput = TRUE)
```

```
## clang++ -mmacosx-version-min=10.13 -std=gnu++11 -I"/Library/Frameworks/R.framework/Resources/include
## clang++ -mmacosx-version-min=10.13 -std=gnu++11 -I"/Library/Frameworks/R.framework/Resources/include
## clang++ -mmacosx-version-min=10.13 -std=gnu++11 -dynamiclib -Wl,-headerpad_max_install_names -undefi
## warning: unknown warning option '-Wno-misleading-indentation'; did you mean '-Wno-binding-in-conditi
## 1 warning generated.
## warning: unknown warning option '-Wno-misleading-indentation'; did you mean '-Wno-binding-in-conditi
## 1 warning generated.
```

```
CdemoModel$calculate()
```

```
## [1] 0
```

```
head(CdemoModel$xOde)
```

```
##      [,1]     [,2]       [,3]     [,4] [,5]
## [1,]    0 508.2000 0.00000000 1.000000    0
## [2,]    1 507.9851 0.03792684 1.176998    0
## [3,]    2 507.7255 0.04608551 1.428389    0
## [4,]    3 507.4107 0.05589347 1.733386    0
## [5,]    4 507.0290 0.06777170 2.103246    0
## [6,]    5 506.5662 0.08214942 2.551640    0
```

test avec deSolve:

```
y <- c(508.2, 0, 1, 0) # intial values
times <- seq(from = 0, to = 50, by = 1) # time sequence
params <- c(0.2, 6, 0, 508.2) # beta, zeta, alpha, Sinit
shaun <- function(t, y, parms){
  dy1 <- - parms[1] * y[1] * y[3] / parms[4] # S
  dy2 <- parms[1] * y[1] * y[3] / parms[4] - parms[2] * y[2] # E
  dy3 <- parms[2] * y[2] - parms[3] * y[1] * y[3] # Z
  dy4 <- parms[3] * y[1] * y[3] # R
  return(list(c(dy1, dy2, dy3, dy4)))
}
ode_nativeR <- deSolve::ode(y, times, shaun, params)[, -1]
head(ode_nativeR)
```

```
##             1          2        3 4
## [1,] 508.2000 0.00000000 1.000000 0
## [2,] 507.9851 0.03792698 1.176998 0
## [3,] 507.7255 0.04608553 1.428391 0
## [4,] 507.4107 0.05589356 1.733388 0
## [5,] 507.0290 0.06777190 2.103251 0
## [6,] 506.5662 0.08214955 2.551646 0
```

Compare with native R:

```
sum(CdemoModel$xOde[,-1]-ode_nativeR)
```

```
## [1] -2.041563e-11
```

It works, awesome !

Add gaussian noise

Now let us have a look to a system with some observation error, which we assume is gaussian.

First, we read in the data from *Shaun of the Dead*:

```
tgrid <- c(0.00, 3.00, 5.00, 6.00, 8.00, 10.00, 22.00, 22.20, 22.50, 24.00, 25.50,
           26.00, 26.50, 27.50, 27.75, 28.50, 29.00, 29.50, 31.50)
zombies <- c(0, 1, 2, 2, 3, 3, 4, 6, 2, 3, 5, 12, 15, 25, 37, 25, 65, 80, 100)
```

Now the code:

```
code <- nimbleCode({

  # system of ODEs
  xOde[1:ngrid, 1:(ndim+1)] <- myModel(y[1:ndim], times[1:ngrid], 0.1, params[1:ndim])
  # priors on parameters
  params[1] ~ dunif(0, 10) # beta
  params[2] ~ dunif(0, 1) # zeta
  params[3] ~ dunif(0, 0.01) # alpha
  params[4] ~ dunif(300, 600) # Sinit

  # observation error
  for (i in 1:ngrid){
    obs_x[i] ~ dnorm(xOde[i, 4], tau.x)
  }

  # prior on error sd
  tau.x <- 1 / var.x
  var.x <- 1 / (sd.x * sd.x)
  sd.x ~ dunif(0, 5)
})
```

Specify the constants, data and initial values:

```
# constants
constants <- list(ngrid = 19,
                  ndim = 4)

# data (pass times and y in constants?)
data <- list(times = tgrid,
             obs_x = zombies,
             y = c(508.2, 0, 1, 0))

# initial values
inits <- list(sdx = 2)
```

Get ready:

```
Rmodel <- nimbleModel(code, constants, data, inits)
# Rmodel$calculate()   ## NA...
conf <- configureMCMC(Rmodel)
```

```
## ===== Monitors =====
## thin = 1: params, sd.x
## ===== Samplers =====
## RW sampler (5)
##   - params[]  (4 elements)
##   - sd.x
```

```
conf$printMonitors()
```

```
## thin = 1: params, sd.x
```

```
conf$printSamplers(byType = TRUE)
```

```
## RW sampler (5)
##   - params[]  (4 elements)
##   - sd.x
```

```
Rmcmc <- buildMCMC(conf)
Cmodel <- compileNimble(Rmodel)
Cmcmc <- compileNimble(Rmcmc, project = Rmodel)
```

Unleash the beast:

```
samplesList <- runMCMC(Cmcmc, 5000,
                       nburnin = 1000,
                       thin = 10,
                       nchains = 2,
                       samplesAsCodaMCMC = TRUE)
```

```
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
```

Check out convergence:

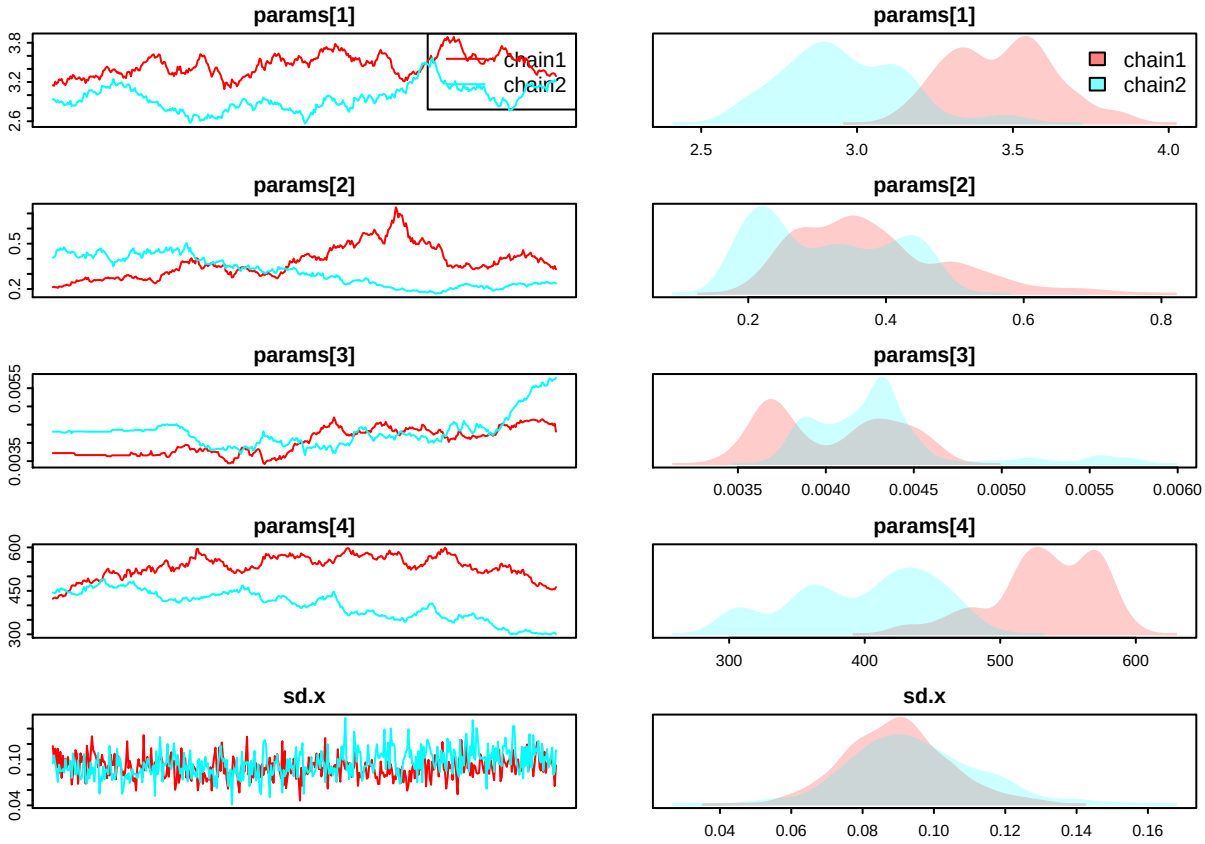
```
library(coda)
gelman.diag(samplesList)
```

```
## Potential scale reduction factors:
##
##           Point est. Upper C.I.
## params[1]       3.72       8.15
## params[2]       4.11      14.81
```

```
## params[3]       1.23       1.41
## params[4]       6.35      13.99
## sd.x            1.30       2.00
##
## Multivariate psrf
##
## 5.49
```

Visualize traceplots and posterior distributions:

```
library(basicMCMCplots)
chainsPlot(samplesList)
```



Apart from the standard deviation of the observation error, the mixing is poor. This is what I had with `OpenBUGS` too, see here.

Can we do something about that? Hopefully yes, and this is what's great with `Nimble`, you have full control of the underlying MCMC machinery. There are useful advices on the Nimble forum here. One reason for poor mixing is correlation in parameters. Let's have a look then.

```
cor(samplesList$chain1)
```

```
##             params[1]  params[2]    params[3]  params[4]          sd.x
## params[1]  1.00000000  0.2569223 0.4472007074  0.5926234 -0.0169919134
## params[2]  0.25692231  1.0000000 0.7243505706  0.5653501 -0.1952381114
## params[3]  0.44720071  0.7243506 1.0000000000  0.1408404  0.0004177985
## params[4]  0.59262337  0.5653501 0.1408404231  1.0000000 -0.2491876288
## sd.x      -0.01699191 -0.1952381 0.0004177985 -0.2491876  1.0000000000
```

```
cor(samplesList$chain2)
```

```
##            params[1]  params[2]  params[3]  params[4]       sd.x
## params[1]  1.0000000 -0.4733005  0.3540405 -0.2517364  0.1741130
## params[2] -0.4733005  1.0000000 -0.1632002  0.7979758 -0.3770379
## params[3]  0.3540405 -0.1632002  1.0000000 -0.5973614  0.2113036
## params[4] -0.2517364  0.7979758 -0.5973614  1.0000000 -0.4252259
## sd.x       0.1741130 -0.3770379  0.2113036 -0.4252259  1.0000000
```

Seems like some parameters are slightly correlated, like β (params[1]) and ζ (params[2]) for example. For the sake of example, let's use block sampling to try and improve mixing.

First, ask what are the samples used currently:

```
conf$printSamplers()
```

```
## [1] RW sampler: params[1]
## [2] RW sampler: params[2]
## [3] RW sampler: params[3]
## [4] RW sampler: params[4]
## [5] RW sampler: sd.x
```

OK, now remove the default samplers for params[1] and params[3] and use a block random walk instead:

```
conf$removeSamplers(c('params[1]', 'params[2]'))
conf$printSamplers()
```

```
## [1] RW sampler: params[3]
## [2] RW sampler: params[4]
## [3] RW sampler: sd.x
```

```
conf$addSampler(target = c('params[1]', 'params[2]'),
                type = 'RW_block')
conf$printSamplers()
```

```
## [1] RW sampler: params[3]
## [2] RW sampler: params[4]
## [3] RW sampler: sd.x
## [4] RW_block sampler: params[1], params[2]
```

Now rebuild, recompile:

```
Rmcmc <- buildMCMC(conf)
Cmodel <- compileNimble(Rmodel)
Cmcmc <- compileNimble(Rmcmc, project = Rmodel)
```

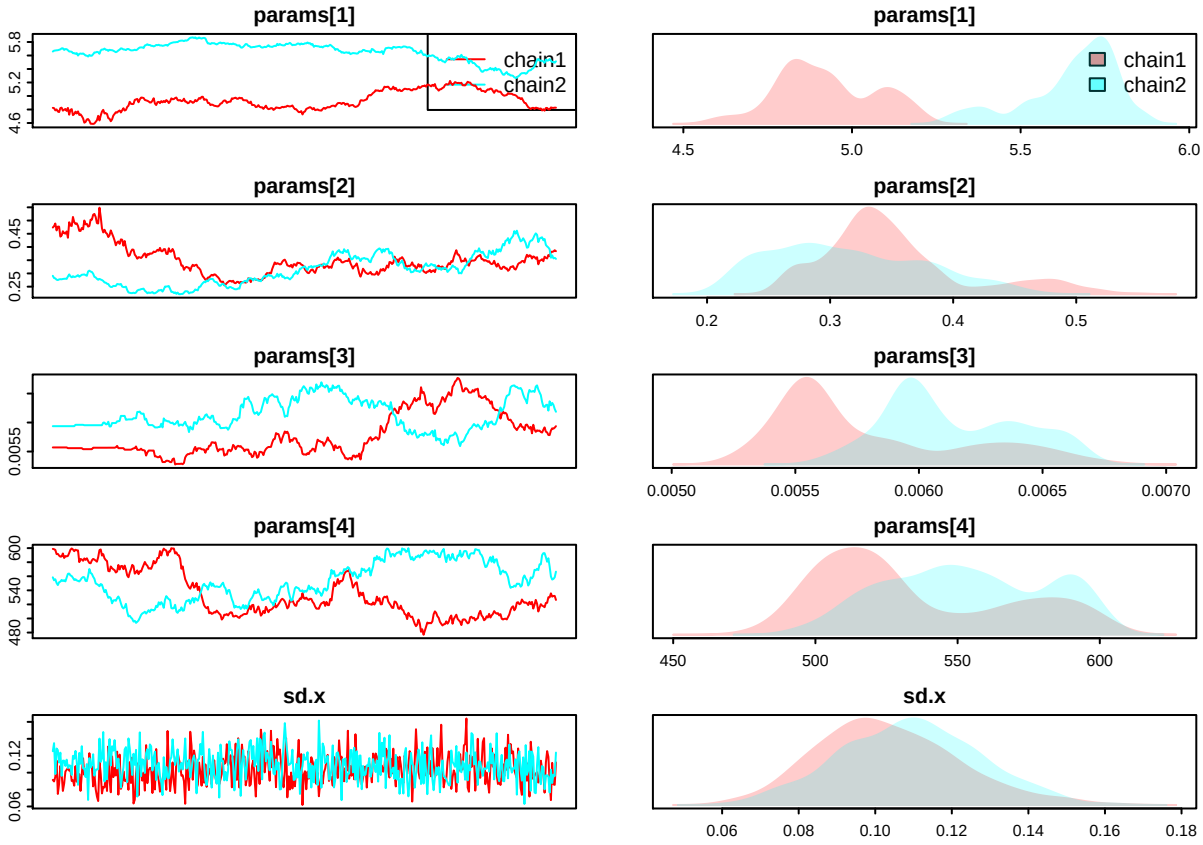
Now rerun:

```
samplesList <- runMCMC(Cmcmc, 5000,
                       nburnin = 1000,
                       thin = 10,
                       nchains = 2,
                       samplesAsCodaMCMC = TRUE)
```

```
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
```

Visualize traceplots and posterior distributions:

```
chainsPlot(samplesList)
```



What if we use another sampler?

```
Rmodel <- nimbleModel(code, constants, data, inits)
conf <- configureMCMC(Rmodel)
```

```
## ===== Monitors =====
## thin = 1: params, sd.x
## ===== Samplers =====
## RW sampler (5)
##   - params[]  (4 elements)
##   - sd.x
```

```
conf$printSamplers()
```

```
## [1] RW sampler: params[1]
## [2] RW sampler: params[2]
## [3] RW sampler: params[3]
## [4] RW sampler: params[4]
## [5] RW sampler: sd.x
```

```
conf <- configureMCMC(Rmodel, onlySlice = TRUE)
```

```
## ===== Monitors =====
## thin = 1: params, sd.x
## ===== Samplers =====
## slice sampler (5)
##   - params[]  (4 elements)
##   - sd.x
```

```
conf$printSamplers()
```

```
## [1] slice sampler: params[1]
## [2] slice sampler: params[2]
## [3] slice sampler: params[3]
## [4] slice sampler: params[4]
## [5] slice sampler: sd.x
```

Now rebuild, recompile:

```
Rmcmc <- buildMCMC(conf)
Cmodel <- compileNimble(Rmodel)
Cmcmc <- compileNimble(Rmcmc, project = Rmodel)
```

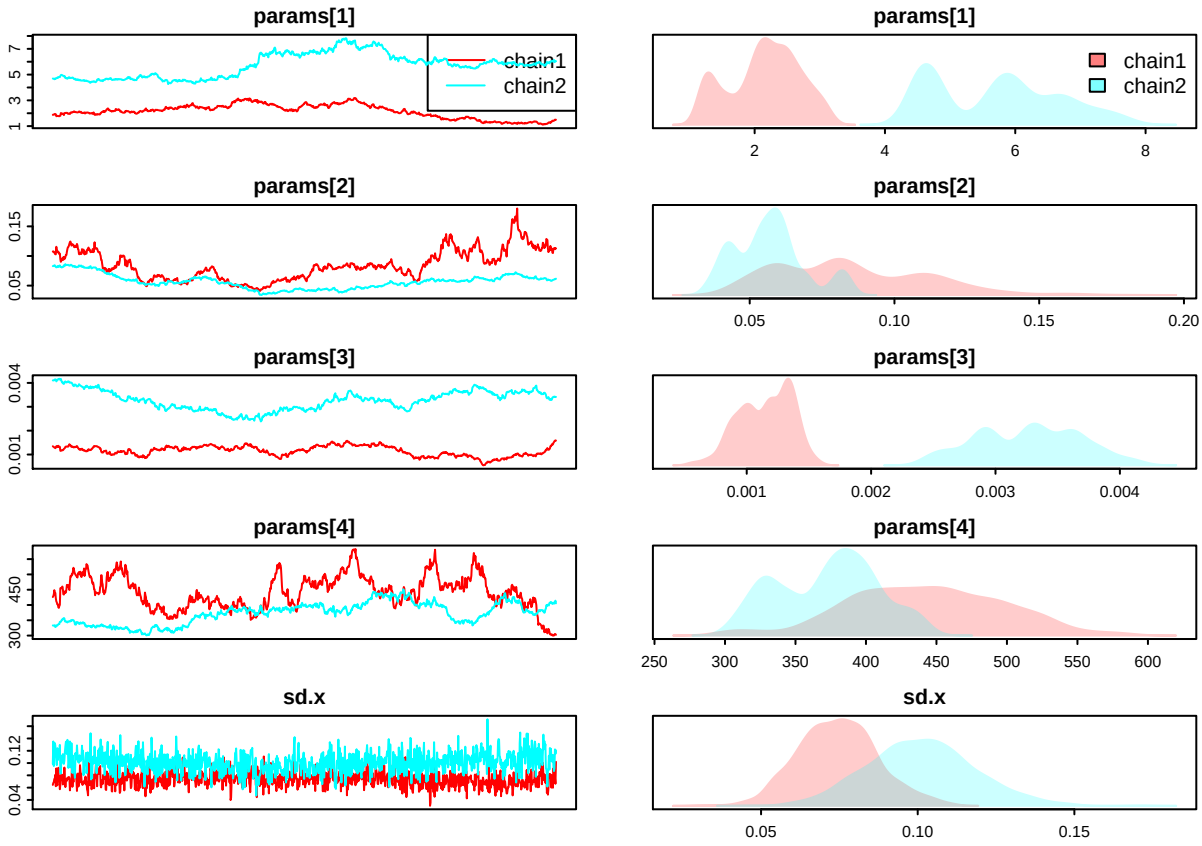
Now rerun:

```
samplesList <- runMCMC(Cmcmc, 2000,
                       nburnin = 1000,
                       thin = 1,
                       nchains = 2,
                       samplesAsCodaMCMC = TRUE)
```

```
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
```

Visualize traceplots and posterior distributions:

```
chainsPlot(samplesList)
```



Much slower.

R version used

```
sessionInfo()
```

```
## R version 4.0.2 (2020-06-22)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS Catalina 10.15.7
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRblas.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRlapack.dylib
##
## locale:
## [1] fr_FR.UTF-8/fr_FR.UTF-8/fr_FR.UTF-8/C/fr_FR.UTF-8/fr_FR.UTF-8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] basicMCMCplots_0.2.5 coda_0.19-4          Rcpp_1.0.5
## [4] odeintr_1.7.1        nimble_0.9.1
##
## loaded via a namespace (and not attached):
```

```
##  [1] codetools_0.2-16 lattice_0.20-41  digest_0.6.27    BH_1.72.0-3
##  [5] R6_2.5.0         grid_4.0.2       magrittr_2.0.1   evaluate_0.14
##  [9] rlang_0.4.9      stringi_1.5.3    rmarkdown_2.5    deSolve_1.28
## [13] tools_4.0.2      stringr_1.4.0    igraph_1.2.6     parallel_4.0.2
## [17] xfun_0.19        yaml_2.2.1       compiler_4.0.2   pkgconfig_2.0.3
## [21] htmltools_0.5.0  knitr_1.30
```