# Supplementary Information Three: Sequential Monte Carlo methods for data assimilation problems in ecology

Kwaku Peprah Adjei[1,2], Rob Cooke[3], Nick Isaac[3], Robert B. O'Hara[1,2]

[1]Department of Mathematical Sciences, Norwegian University of Science and Technology, Trondheim, Norway

[2]Centre of Biodiversity Dynamics, Norwegian University of Science and Technology, Trondheim, Norway

[3]Center of Ecology and Hydrology, Wallingford, UK

## 1 Introduction

Ecological time series data are collected to explain spatio-temporal changes in species distribution. Species distribution models that explain the abundance or occupancy of the species need to be updated to reflect the current trends in the species distributions.

In recent years, the state space models (SSMs) have become widely used in analyzing such data. These SSMs have been fitted in the Bayesian framework using the Markov chain Monte Carlo (MCMC) approach. This approach can be very computationally expensive and takes a relatively long time to fit. Fitting the SSMs with this approach can be unfeasible in terms of computational load and time needed to run model which increases with data size.

A faster approach to fitting the SSMs is using the sequential Monte Carlo (SMC) approach. The SMC uses sequential importance sampling (SIS) to obtain importance weights that are used to generate posterior distributions of latent states at every time step and also update the other parameters in the model (using the particle MCMC approach; Andrieu, Doucet, and Holenstein (2010)). The SMC approaches have been implemented in packages such as `nimbleSMC` (Michaud et al. 2021), and this package will be referred to in this document.

This document seeks to demonstrate how to use information in an already-fitted SSM using particle MCMC approaches to estimate the posterior distribution of the latent states and model parameters. The demonstration applies the proposed framework in the main paper to a simulated dataset from the model described in the main paper (section 3.1). It is worth noting the choice of the true values of model parameters and number of particles used is the same as described in the main paper.

The vignette assumes that the reader is familiar with the `nimbleSMC` R-package and its functions (the reader is referred to Chapter 8 of de Valpine et al. (2022) and Michaud et al. (2021) for details on how to fit SSMs using SMC approach in NIMBLE (Valpine et al. 2017)). The rest of the vignette is organised as follows: the

set up for the analysis are described, then the reduced SSM are fitted using the MCMC approaches with JAGS (Depaoli, Clifton, and Cobb 2016) and NIMBLE (Valpine et al. 2017) and then demonstrate how to process the stored posterior samples from the reduced model in a format that can be used for th updating process. The final part of this vignette presents the updated model and summary plots from all the fitted models.

The proposed model framework can be accessed through the functions written in this package `nimMCMCSMCupdates` which can be installed as follows:

```
devtools::install_github("Peprah94/nimMCMCSMCupdates")
```

## 2 Set-up for analysis

To begin with, the relevant packages are loaded and data is simulated. The following packages are to be loaded for this document to run.

```
library(nimMCMCSMCupdates)
library(nimble)
library(nimbleSMC)
library(dplyr)
library(ggplot2)
library(ggmcmc)
library(kableExtra)
library(rjags)
library(coda)
library(mcmcse)
```

### 2.1 Simulating data

Data is simulated from the linear Gaussian state space model for $t = 50$ years, $a = 0.5$, $c = 1$.

```
#seed seed for reproduceable results
set.seed(1)


nyears = 50 #Number of years
iNodePrev <- 45 # The number of years for reduced model


# function to simulate the data
sim2 <- function(a, b, c, t){
  x <- y <- numeric(t)
  x[1] <- rnorm(1, a, 1 )
```

```
  y[1] <- rnorm(1, c*x[1], 1)
  #eta <- rnorm(t, 0, b)
  for(k in 2:t){
    x[k] <- rnorm(1, (a*x[k -1]) , 1)
    y[k] <- rnorm(1, x[k]*c, 1)
  }
  return(list(x=x, y=y))
}

# simulating the data
simData <- sim2(a = 0.5,
                # b = 0.1,
                 c = 1,
                 t = nyears)

#save data
save(simData, file = "simData.RData")

# Taking a look at the data
str(simData)
```

```
List of 2
 $ x: num [1:50] -0.126 -0.899 -0.12 0.427 0.79 ...
 $ y: num [1:50] 0.0572 0.6964 -0.9404 1.1658 0.4841 ...
```

## 2.2 General parameters

Some common parameters that will be used throughout the process of fitting the reduced SSM are defined. These parameters are the number of MCMC iterations, number of chains, thinning value and the parameters to monitor. For each MCMC or SMC approach to be used in fitting the reduced model, we run the MCMC for 50000 iterations, and use the first 45000 samples as burn-in samples with thinning value of 1.

```
# Taking a look at the data
nIterations = 50000 #Number of iterations
nChains = 3 #number of chains
nBurnin = 45000 #number of burn-in samples
nParFiltRun = 100 # number of particles
parametersToMonitor <- c("a", "c", "x") # parameters to monitor from the MCMC
```

# 3 Fitting reduced SSM

This tutorial will assume that the reduced model is fitted to the observations $y$ from time 1 to 45. The reduced SSM is fitted with MCMC using JAGS (Depaoli, Clifton, and Cobb 2016) and NIMBLE (Valpine et al. 2017). In this section, we show how the linear Gaussian SSM is fitted using the these two approaches and save the posterior distribution of the latent states and model parameters. Specifically, we aim to save the posterior samples of the latent states $x$ and model parameters $a$, and $c$ .

## 3.1 Fitting reduced model with JAGS

We proceed to fit the reduced model with JAGS assuming the reader is familiar with the BUGS language and running MCMC with JAGS (Refer to Depaoli, Clifton, and Cobb (2016) for tutorials on how to fit the model with JAGS).

*Define and set-up JAGS model*

To fit the reduced model with JAGS, we first define our JAGS model using the BUGS syntax. We do this by embedding the BUGS model in R.

```
# defining JAGS model in R
cat("model{
  x[1] ~ dnorm(a, 1)
  y[1] ~ dnorm(c*x[1], 1)
  for(i in 2:t){
    x[i] ~ dnorm(a*x[i-1], 1)
    y[i] ~ dnorm(x[i] * c, 1)
  }

  a ~ dunif(0, 1)
  c ~ dnorm(1,1)
}", file="lgssm.txt"
)

#setting the parameters needed to run the MCMC
data <- list(
  y = simData$y[-c((iNodePrev+1):50)],
  t = iNodePrev
)

# Initial values
```

```
inits <- list(
  a = 0.1,
  c = 1
)
```

We then set up the syntax defined above to run in JAGS. We are going to run the model for 3 chains, with no number of adaptations.

```
jagsModel <- jags.model(file = "lgssm.txt",
                        data = data,
                        inits = inits,
                        n.chains = nChains)
```

```
Compiling model graph
   Resolving undeclared variables
   Allocating nodes
Graph information:
   Observed stochastic nodes: 45
   Unobserved stochastic nodes: 47
   Total graph size: 184

Initializing model
```

*Run the JAGS model and saving posterior samples*

The JAGS model is run and the posterior samples are saved. We run 50000 iterations and use the first 45000 samples as burn-in samples.

```
## run JAGS and save posterior samples
samps <- coda.samples( jagsModel , parametersToMonitor, n.iter= nIterations)
jagsReducedSamples <- window(samps, start=nBurnin+1)
```

## 3.2 Fitting reduced model with NIMBLE

We also assume that the user is familiar with the R-package `nimble` (Valpine et al. 2017) and how to fit models with it (See de Valpine et al. (2022) for details on how to run MCMC with NIMBLE package).

*Defining the NIMBLE model*

We first define our nimble model. This process involves writing the BUGS code, defining the data, constant and initial values components of the the nimbleModel function in `nimble` package.

```
lgSSMCode <- nimbleCode({
  x[1] ~ dnorm(a, 1)
  y[1] ~ dnorm(c*x[1], 1)
  for(i in 2:t){
    x[i] ~ dnorm(x[i-1] * a , 1)
    y[i] ~ dnorm(x[i] * c, 1)
  }

# Priors on model parameters
  a ~ dunif(0, 1)
  c ~ dnorm(1,1)
})

data <- list(
  y = simData$y[-c((iNodePrev+1):50)]
)
# Defining constants
constants <- list(
  t = iNodePrev
)
# Initial values
inits <- list(
  a = 0.1,
  c = 1
)
# Define the nimbleModel with the various components defined
lgSSModel <- nimbleModel(lgSSMCode,
                                data = data,
                                constants = constants,
                                inits = inits,
                                check = FALSE)
```

*Configure, build, compile and run model*

We configure the MCMC by adding the parameters to monitor, build and compile the configured model. The compiled model is then run and the posterior samples are saved.

```
#Compile model
    nMCompile <- compileNimble(lgSSModel)
```

```
Compiling
  [Note] This may take a minute.
  [Note] Use 'showCompilerOutput = TRUE' to see C++ compilation details.
```

```
# configure model
    cMCMC <- configureMCMC(lgSSModel, monitors = parametersToMonitor)
```

```
===== Monitors =====
thin = 1: a, c, x
===== Samplers =====
RW sampler (1)
  - a
conjugate sampler (46)
  - c
  - x[]  (45 elements)
```

```
    # block both a and c
    cMCMC$removeSampler(c("a", "c"))
    cMCMC$addSampler(target = c("a", "c"), type = "RW_block")
```

```
  [Note] Assigning an RW_block sampler to nodes with very different scales can result in low MCMC effic
```

```
    #build model
    bMCMC <- buildMCMC(cMCMC)


  # Compile the model to run in c++
    coMCMC <- compileNimble(bMCMC, project = nMCompile)
```

```
Compiling
  [Note] This may take a minute.
  [Note] Use 'showCompilerOutput = TRUE' to see C++ compilation details.
```

```
    #runMCMC
    mcmc.out <- runMCMC(coMCMC,
                        niter = nIterations,
```

```
                              nchains = nChains,
                              nburnin = nBurnin,
                              setSeed = TRUE,
                              samples=TRUE,
                              samplesAsCodaMCMC = TRUE,
                              summary = TRUE,
                              WAIC = FALSE)
```

```
running chain 1...

|-------------|-------------|-------------|-------------|
|-------------------------------------------------------|

running chain 2...

|-------------|-------------|-------------|-------------|
|-------------------------------------------------------|

running chain 3...

|-------------|-------------|-------------|-------------|
|-------------------------------------------------------|
```

```
    #save posterior samples
nimbleReducedSamples <- mcmc.out$samples
```

### 3.3 Saving all samples

We save all the results from the two fitted reduced SSMs in a list to be used in the updating process.

```
mcmcSamplesList <- list(jagsReducedSamples = jagsReducedSamples,
                        nimbleReducedSamples = nimbleReducedSamples)
save(mcmcSamplesList,
     file = "mcmcSample.RData")
```

## 4 Converting MCMC output for updating process

We load the saved posterior samples for the updating process. It is worth noting that all the samples are `mcmc.list` class, so the procedure we use to prepare the posterior samples from any of these approaches can be generalized to any MCMC approach of this class.

```
#load saved that
load("mcmcSample.RData")


#Check the class of each of the returned posterior samples
lapply(mcmcSamplesList, function(x) class(x))
```

```
$jagsReducedSamples
[1] "mcmc.list"


$nimbleReducedSamples
[1] "mcmc.list"
```

**Creating model values from saved posterior samples**

The posterior samples need to be stored in as modelValues to be used for updating the SSM (see Chapter 15 of de Valpine et al. (2022) for details on modelValues definition). To do this, we need to re-define the model is we used JAGS as a nimble model, or use our reduced model from the MCMC with `nimble`. In this example, we re-define the JAGS model as a nimble model. We do this by copying the text file into a nimbleCode and then we can define the nimbleModel as we did for the two examples.

```
jagsToNimbleCode <- nimbleCode({
   x[1] ~ dnorm(a, 1)
  y[1] ~ dnorm(c*x[1], 1)
  for(i in 2:t){
    x[i] ~ dnorm(a*x[i-1], 1)
    y[i] ~ dnorm(x[i] * c, 1)
  }


  a ~ dunif(0, 1)
  c ~ dnorm(1,1)
})


data <- list(
  y = simData$y
)


# Defining constants
constants <- list(
  t = iNodePrev
```

```
)

# Initial values
inits <- list(
  a = 0.1,
  c = 1
)

# Define the nimbleModel with the various components defined
jagsToNimbleModel <- nimbleModel(jagsToNimbleCode,
                                 data = data,
                                 constants = constants,
                                 inits = inits,
                                 check = FALSE)

# Define latent states and top-level parameters
latent = "x"
target = c("a", "c")
```

After defining the nimble model, we can create the modelValues object. The modelValues object is created for each MCMC chain. In this example, we show how to do it for the first chain.

```
# Expand the node names for the latent variables
latentNodes <- jagsToNimbleModel$expandNodeNames(latent)

# Find all the latent nodes
# Function in nimbleSMC
nodes <- findLatentNodes(jagsToNimbleModel,
                         latent,
                         timeIndex = 1)

# specify the last node from the reduced model
reducedModel <- lgSSModel #what we used to fit the MCMC reduced model
lastNodes <- findLatentNodes(reducedModel, latent, timeIndex= 1 )
lastNode <- lastNodes[length(lastNodes)]
print(lastNode)
```

```
#find the dimension of the nodes
dims <- lapply(nodes, function(n) nimDim(jagsToNimbleModel[[n]]))
if(length(unique(dims)) > 1)
  stop('sizes or dimensions of latent states varies')

# create a vars vector that contains the names of the latent nodes and top-level parameters
vars <- c(jagsToNimbleModel$getVarNames(nodes =  nodes), target)

#create a list of model objects to store the variable values
modelSymbolObjects <- jagsToNimbleModel$getSymbolTable()$getSymbolObjects()[vars]

# retrun the names, type ans size components for the modelValues.
# These are the essential components needed to create modelValues
names <- sapply(modelSymbolObjects, function(x)return(x$name))
type <- sapply(modelSymbolObjects, function(x)return(x$type))
size <- lapply(modelSymbolObjects, function(x){
  y <- x$size
  #Make sure variables with nDim= 0 will have size of 1
  t <- length(y)
  rr <- c()
  if(t > 1){
rr <- y
  }else{
    if(length(y)>0){
   rr <- y
    }else{
 rr <- 1}
  }
  return(rr)
  }
)

#create ModelValues object
mvSamplesEst <- modelValues(modelValuesConf(vars = names,
                                            types = type,
                                            sizes = size))
```

```
#let's check the mvSamplesEst
print(mvSamplesEst)

# The modelValue object created has size of 1.
mvSamplesEst$getSize()

# resize the modelValues object created to be equal to (number of iterations - burnin samples)/n.thin
mcmcSampleSize = (nIterations - nBurnin)
resize(mvSamplesEst, mcmcSampleSize)

# check size now
mvSamplesEst$getSize()

# retrieve posterior samples for the first chain
mcmcOut <- mcmcSamplesList$jagsReducedSamples[[1]]
model <- jagsToNimbleModel

#create mvEst for the first chain
for(iter in 1:mcmcSampleSize){
  for(j in 1:length(names)){
      if(names[j] == latent){
      estValues <- c(mcmcOut[iter, reducedModel$expandNodeNames(lastNode)],
                     rep(0, length(model$expandNodeNames(names[j]))-1))
      names(estValues) <- model$expandNodeNames(names[j])
      }else{
        namesExpanded <- reducedModel$expandNodeNames(names[j])
        lastName <- namesExpanded[length(namesExpanded)]
        extraVals <- values(model, (model$expandNodeNames(names[j]))[-1])
       estValues <- c(mcmcOut[iter, lastName ], extraVals)
       names(estValues) <- model$expandNodeNames(names[j])
      }
          mvSamplesEst[[names[j]]][[iter]] <-  estValues
  }
}
```

This process of converting MCMC posterior samples to modelValues that can be used by `nimbleSMC` is written

in **updateUtils** function in our `nimbleMCMCSMCupdates` package. It takes as input the reduced model, the MCMC samples, latent state, target vectors, the MCMC sample size, the timeIndices for the **findLatentNodes** function. The code below show produce the same results as above.

```
    updateVars <- updateUtils(model = jagsToNimbleModel, #model to fit
                              reducedModel = lgSSModel, #reduced model
                mcmcOut = mcmcSamplesList$jagsReducedSamples[[1]], #MCMC samples
                          latent = latent, #latent state
               target = target, #model parameters to update
               iNodeAll = TRUE, #logic value indicating whether you need to save all the MCMC values
               n.iter = nIterations - nBurnin, #number of iterations
               m = nParFiltRun, #number of particles
               timeIndex = 1 #time index
               )
```

```
Saving unsampled and sampled values in model values for updating
```

# 5 Updating process using particle MCMC

After saving posterior samples as modelValues, we then proceed to fit the updated SSM. We modified the bootstrap and auxiliary particle filter algorithms as well as the MCMC samplers in the `nimbleSMC` package. We use the modified functions, which can be found in our `nimbleMCMCSMC` package, to illustrate the updating process. Note that the entire process defined here has to be repeated for number of chains times. We illustrate how to use the results from JAGS for the updating process.

**Re-defining the nimbleModel**

The reduced SSM needs to be updated with the observation from time $t = 45$ to $t = 50$. To do this, we change the data and constant components of our reduced model and create a new nimble model.

```
data <- list(
  y = simData$y
)
# Defining constants
constants <- list(
  t = nyears
)
# Initial values
inits <- list(
  a = 0.1,
  c = 1
```

```
)
# Define the nimbleModel with the various components defined
lgSSMUpdated <- nimbleModel(lgSSMCode,
                            data = data,
                            constants = constants,
                            inits = inits,
                            check = FALSE)
```

**Build updated particle filter and set up MCMC configuration**

```
  modelMCMCconf <- nimble::configureMCMC(lgSSMUpdated,
                                         nodes = NULL,
                                         monitors = parametersToMonitor)
```

```
===== Monitors =====
thin = 1: a, c, x
===== Samplers =====
(no samplers assigned)
===== Comments =====
```

```
# Define control element
  pfControl = list(saveAll = TRUE,
                   smoothing = FALSE,
                   M = nyears - iNodePrev,
                   iNodePrev = iNodePrev)

# Build particle filter
particleFilter <- nimMCMCSMCupdates::buildBootstrapFilterUpdate(model = lgSSMUpdated ,
nodes = latent,
mvSamplesEst = updateVars,
target = target,
control = pfControl)
```

**Build, compile and run MCMC**

The modified Metropolis-Hastings random walk block sampler is used to sample the model parameters. The sampler is called **RW_PF_blockUpdate**.

```
pfTypeUpdate = 'bootstrapUpdate'

#Add RW_blockUpdate sampler
```

```
modelMCMCconf$addSampler(target = target,
                         type = 'RW_PF_blockUpdate',
                         control = list(latents = latent,
                                        target = target,
                                        adaptInterval = mcmcSampleSize,
                                        pfControl = list(saveAll = TRUE,
                                                         M = nyears - iNodePrev,
                                                         iNodePrev = 1),
                                        pfNparticles = nParFiltRun,
                                        pfType = pfTypeUpdate,
                                        extraVars = NULL,
                                        mvSamplesEst = mvSamplesEst,
                                        logLikeVals = NA))

#Check if we are getting the varaiables we want to monitor
modelMCMCconf$printMonitors()

#build MCMC
  modelMCMC <- buildMCMC(modelMCMCconf)

  #compile MCMC
compiledList <- nimble::compileNimble(lgSSMUpdated,
                                      modelMCMC,
                                      resetFunctions = TRUE)

#run MCMC
mcmc.out <- nimble::runMCMC(compiledList$modelMCMC,
                            niter = nIterations - nBurnin,
                            nchains = 1,
                            nburnin = 0,
                            setSeed = TRUE,
                            samples=TRUE,
                            samplesAsCodaMCMC = TRUE,
                            summary = TRUE,
                            WAIC = FALSE)
```

```
  #save output
  save(mcmc.out, file = "lgssmUpdatedResults.RData")



  head(mcmc.out$summary)
```

The entire process described above is written up in the **spartaNimUpdates** function in our `nimMCMCSMCupdates` package. We use this function to run the updated model for all the SSM fitted with JAGS.

```
# Replicate the updated and reduced models
newModelReduced <- jagsToNimbleModel$newModel(replicate = TRUE)
newModelUpdated <- lgSSMUpdated$newModel(replicate = TRUE)

#reduced model list
mcmcList <- list()
mcmcList$samples <- mcmcSamplesList$jagsReducedSamples
mcmcList$summary <- NULL

lgSSMUpdatedResults <- nimMCMCSMCupdates::spartaNimUpdates(
  model = newModelUpdated, #nimble model
  reducedModel = newModelReduced, # reduced nimbleModel
  latent = "x", #latent variable
  pfType = "bootstrap", # type of SMC
  MCMCconfiguration = list(target = c('a','c'), #top-level parameters
                           additionalPars = "x", #additional parameters to monitor
                           n.iter = nIterations - nBurnin, #number of iterations
                           n.chains = nChains, # number of chains
                           n.burnin = 0, # number of burnin samples
                           n.thin = 1),  # number of thinning samples
  postReducedMCMC = mcmcList,# MCMC posterior samples from reduced SSM
  pfControl = list(saveAll = TRUE,
                   smoothing = FALSE,
                   M = nyears - iNodePrev,
                   iNodePrev = iNodePrev)
)
```

```
save(lgSSMUpdatedResults, file = "lgSSMUpdatedResults1.RData")
```

The **spartaNimUpdates** function returns the posterior samples, summary of the posterior samples and the run time of the fitted model.

## 6 Comparing the various models fit

As a final part of this document, we present figures of the posterior mean and Monte Carlo standard error (MCSE) of latent states in Figure 1, and the convergence of model parameters estimated from the reduced model using JAGS and NIMBLE in Figure 2 and Figure 3 respectively and the updated model in Figure 4.

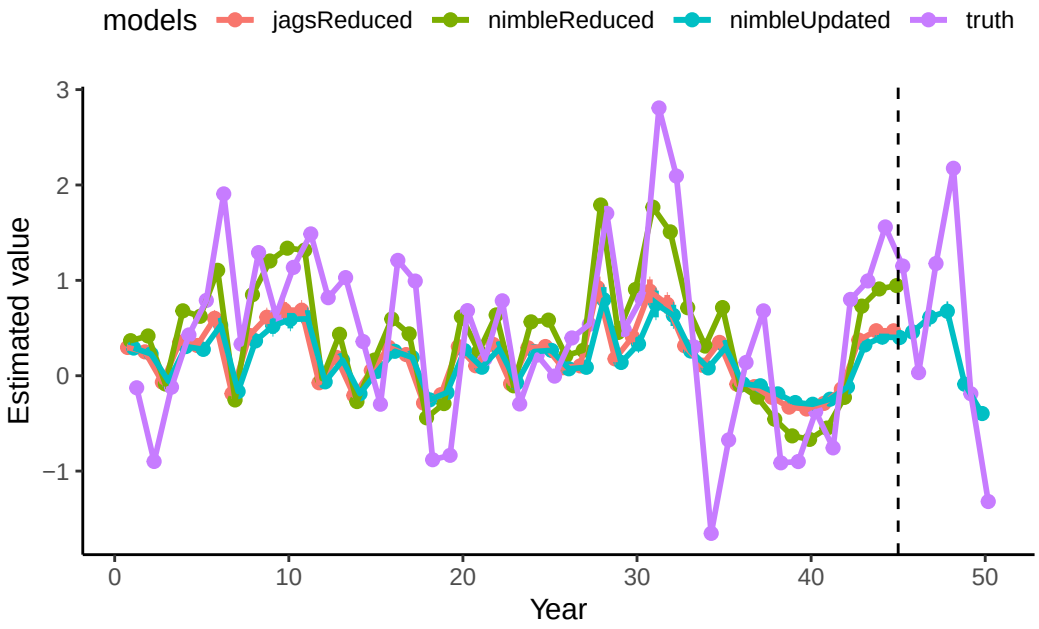


Figure 1: Posterior mean (± MCSE) of the latent state distribution estimated from SSMs fitted with JACS, NIMBLE and updated model.

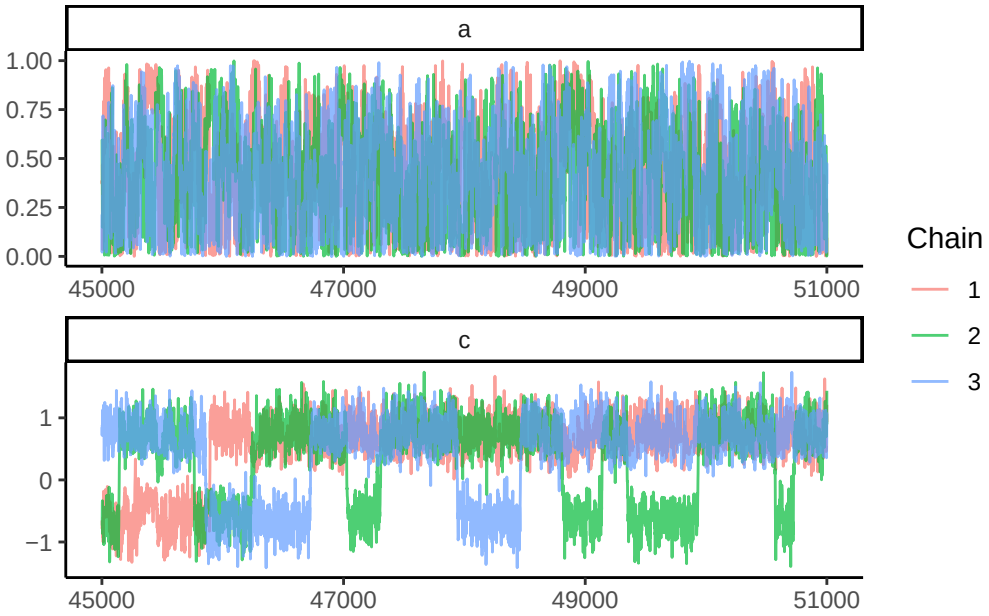


Figure 2: Traceplot of MCMC chains from the reduced model fitted with JAGS.

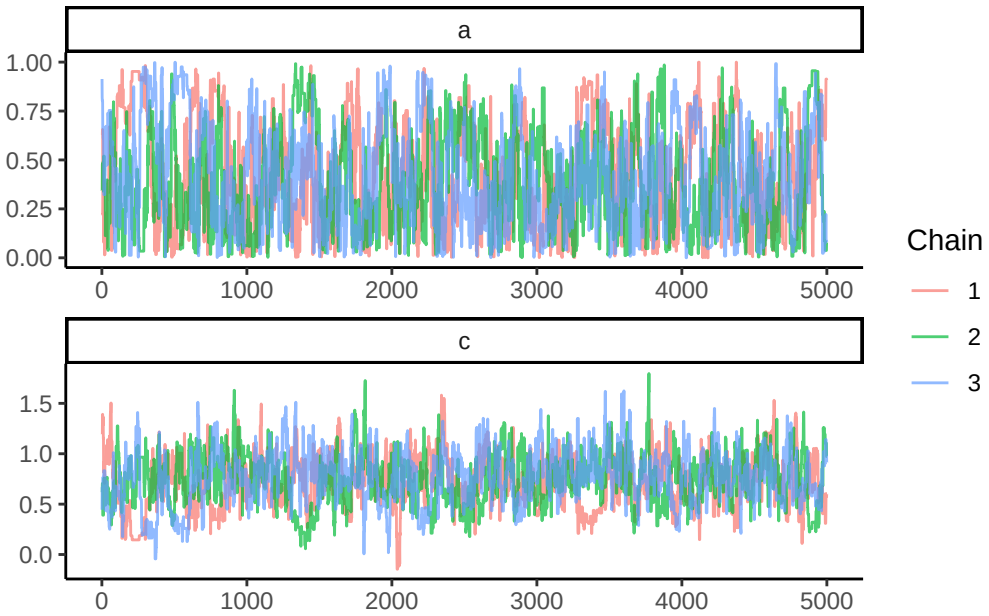


Figure 3: Traceplot of MCMC chains from the reduced model fitted with NIMBLE.

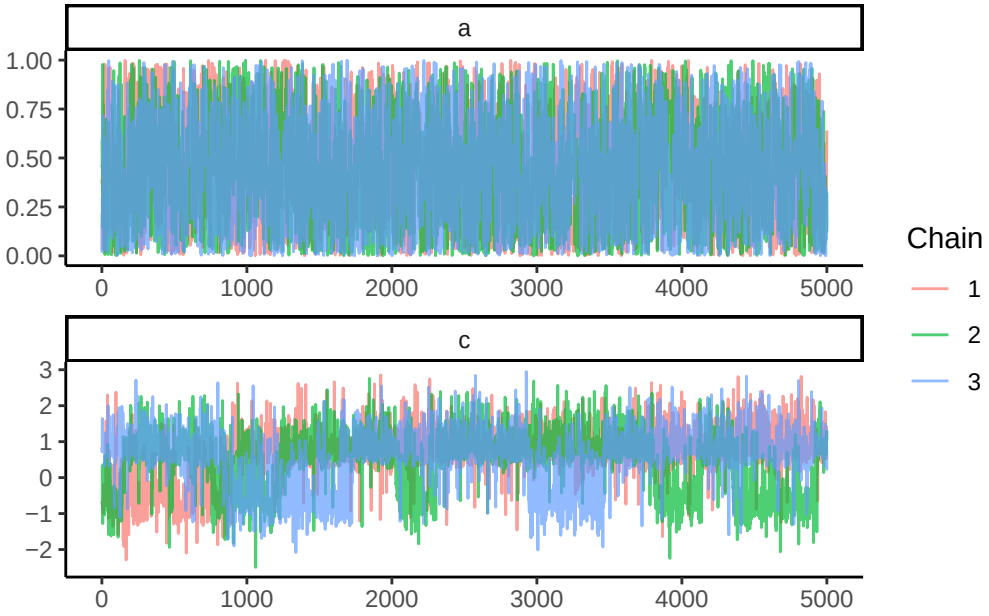


Figure 4: Traceplot of MCMC chains from the updated model.

# 7 References

Andrieu, Christophe, Arnaud Doucet, and Roman Holenstein. 2010. "Particle Markov Chain Monte Carlo Methods." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 72 (3): 269–342.

de Valpine, Perry, Christopher Paciorek, Daniel Turek, Nick Michaud, Cliff Anderson-Bergman, Fritz Obermeyer, Claudia Wehrhahn Cortes, Abel Rodrìguez, Duncan Temple Lang, and Sally Paganin. 2022. *NIMBLE User Manual* (version 0.13.1). https://doi.org/10.5281/zenodo.1211190.

Depaoli, Sarah, James P Clifton, and Patrice R Cobb. 2016. "Just Another Gibbs Sampler (JAGS) Flexible Software for MCMC Implementation." *Journal of Educational and Behavioral Statistics* 41 (6): 628–49.

Michaud, Nicholas, Perry de Valpine, Daniel Turek, Christopher J Paciorek, and Dao Nguyen. 2021. "Sequential Monte Carlo Methods in the Nimble and nimbleSMC r Packages." *Journal of Statistical Software* 100: 1–39.

Valpine, Perry de, Daniel Turek, Christopher J Paciorek, Clifford Anderson-Bergman, Duncan Temple Lang, and Rastislav Bodik. 2017. "Programming with Models: Writing Statistical Algorithms for General Model Structures with NIMBLE." *Journal of Computational and Graphical Statistics* 26 (2): 403–13.