Vignette for sequential Monte Carlo methods for data assimilation problems

Kwaku Peprah Adjei[1,2,3]   Robert B. O'Hara[4]

Table of contents

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 and the model parameters updated to reflect the current trends in the species.

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 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 once can use MCMC models already fitted to the SSMs and update them using the SMC approach. The bootstrap and auxiliary PFs were discussed in the main paper, but this document will focus on bootstrap particle filter. The reader is expected to be familiar with the `nimbleSMC` package and its functionalities (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). The first part provides a brief introduction to the state space models (SSMs), sequential Monte Carlo (SMC) approaches (specifically the bootstrap particle filter) and the particle MCMC with the necessary changes made to accomodate the updating process when new streams of data are obtained. The next part shows how to fit the model to the simulated data used in Example 1 in the main paper.

2 Examples

In this document, we demonstrate the use of our proposed model framework by applying them to a simulated dataset. The data was simulated from the linear Gaussian state space model in Example 1 of our main paper. The various functions and changes made to the `nimbleSMC` package filtering algorithms and samplers are written in our package `nimMCMCSMCupdates`. The `nimMCMCSMCupdates` package can be installed from github as follows:

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

3 Set-up

Simulating data and loading packages

3.1 Load packages

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)
```

3.2 Simulating data

We simulate data from the linear Gaussian state space model for $t = 50$ years.

```
#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, mu0){
  x <- y <- numeric(t)
  x[1] <- rnorm(1, mu0, 1 )
  y[1] <- rnorm(1, x[1], 1)

  for(k in 2:t){
    x[k] <- rnorm(1, a*x[k -1] + b, 1)
    y[k] <- rnorm(1, x[k-1]*c, 1)
```

```
  }
  return(list(x=x, y=y))
}

# simulating the data
simData <- sim2(a = 0.8,
                b = 1,
                c = 1.5,
                t = nyears,
                mu0 = 0.2)
# Taking a look at the data
str(simData)
```

```
List of 2
 $ x: num [1:50] -0.426 -0.177 1.188 2.438 3.526 ...
 $ y: num [1:50] -0.243 0.956 -1.086 2.52 3.351 ...
```

4 Fitting reduced SSM

This tutorial will assumes that the reduced model is fitted to the observations y from time $t = 1$ to $t = 45$. The reduced SSM is fitted with MCMC using JAGS [**cite**] and NIMBLE [**cite**] and with particle MCMC using [**cite**]. In this section, we show how the linear Gaussian state space model is fitted using the these three approaches and the results we need for the updating process. Specifically, we aim at saving the posterior samples of the latent states x and top-level parameters a, b, c and μ_0.

We proceed by first defining some common parameters that will be used throughout the process of fitting the reduced SSM. 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 1000 iterations, and use the first 500 samples as burn-in samples with thinning value of 1.

```
nIterations = 10000
nChains = 3
nBurnin = 5000
nParFiltRun = 1000
parametersToMonitor <- c("a", "b", "c", "mu0", "x")
```

4.1 Fitting reduced model with JAGS

We proceed to fit the reduced model with JAGS assuming the reader is familiar with the BUGS language and possible running MCMC with JAGS (Refer to **citation** 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(mu0, 1)
  y[1] ~ dnorm(x[1], 1)
  for(i in 2:t){
    x[i] ~ dnorm(x[i-1] * a + b, 1)
    y[i] ~ dnorm(x[i] * c, 1)
  }
  a ~ dunif(0, 1)
  b ~ dnorm(0, 1)
```

```
  c ~ dnorm(1,1)
  mu0 ~ dnorm(0, 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,
  b = 0,
  mu0= 0.2,
  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: 49
   Total graph size: 229

Initializing model
```

Run the JAGS model and saving posterior samples

The JAGS model is run and the posterior samples are saved. We run 1000 iterations and use the first 500 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)
```

4.2 Fitting reduced model with NIMBLE

We also assume that the user is familiar with the NIMBLE package [**cite**] and how to fit models with it (See [**cite**] for details on how to rum MCMC with NIMBLE package [**cite**]).

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. **What does the nimbleModel function do?**

```
lgSSMCode <- nimbleCode({
  x[1] ~ dnorm(mu0, 1)
  y[1] ~ dnorm(x[1], 1)
```

```
  for(i in 2:t){
    x[i] ~ dnorm(x[i-1] * a + b, 1)
    y[i] ~ dnorm(x[i] * c, 1)
  }
  a ~ dunif(0, 1)
  b ~ dnorm(0, 1)
  c ~ dnorm(1,1)
  mu0 ~ dnorm(0, 1)
})


data <- list(
  y = simData$y[-c((iNodePrev+1):50)]
)
# Defining constants
constants <- list(
  t = iNodePrev
)
# Initial values
inits <- list(
  a = 0.1,
  b = 0,
  mu0= 0.2,
  c = 1
)
# Define the nimbleModel with the various components defined
lgSSModel <- nimbleModel(lgSSMCode,
```

```r
                                    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.

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


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


    #build model
    bMCMC <- buildMCMC(cMCMC)


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


    #runMCMC
    mcmc.out <- runMCMC(coMCMC,
                              niter = nIterations,
                              nchains = nChains,
                              nburnin = nBurnin,
                              setSeed = TRUE,
```

```
                              samples=TRUE,
                              samplesAsCodaMCMC = TRUE,
                              summary = TRUE,
                              WAIC = FALSE)

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

4.3 Fitting reduced SSM using pMCMC

Lastly, we fit the reduced SSM with particle MCMC **cite** using the `nimbleSMC` [**cite**] package. Similar to the assumptions made for the first two MCMC models fit, we assume that the user is familiar with the `nimbleSMC` package [**cite**] and how to fit models with it (See [**cite**] for details on how to run pMCMC with `nimbleSMC` package [**cite**]).

Defining nimble Model As with the example in the preceeding subsection, we define our nimble model. We use the same nimble model(`lgSSModel`) used in that section, but we need to replicate it since it has been compiled for the MCMC run above.

```
lgSMCmodel <- lgSSModel$newModel(replicate = TRUE)
#compile nimble model
clgSMCmodel <- compileNimble(lgSMCmodel)
```

configure model and Build Particle filter

After defining our nimble model, we first build a particle to be passed on to the MCMC part of the pMCMC. In this example, we build a bootstrap particle filter.

```
mcmcConf <- configureMCMC(lgSMCmodel,
                          nodes = NULL,
                          monitors = parametersToMonitor)
```

```
===== Monitors =====
thin = 1: a, b, c, mu0, x
===== Samplers =====
(no samplers assigned)
===== Comments =====
```

```
particleFilter <- nimbleSMC::buildBootstrapFilter(lgSMCmodel,
                      nodes = "x", #latent state
                      control = list(saveAll = TRUE, #save weights and latent states for e
                                     smoothing = FALSE,
                                     initModel = FALSE))


#compile particle filter
#compileNimble(lgSMCmodel,  particleFilter)
```

Build MCMC part of the pMCMC

We then use the particle filter defined above to configure our MCMC. The configured model is built, compiled and run to obtain the posterior samples. The posterior samples are saved to be used for the updating process.

```
#add sampler for pMCMC
mcmcConf$addSampler(target = c("a", "b", "c", "mu0"),
                    type = "RW_PF_block",
                    pf = particleFilter,
                    adaptive = FALSE,
                    pfNparticles = 1000,
                    latents = "x")
```

```
#mcmcConf$addMonitors("x")
#building MCMC
bMCMCsmc <- buildMCMC(mcmcConf)

# compile MCMC
cMCMCsmc <- compileNimble(bMCMCsmc,
                          project = lgSMCmodel,
                          resetFunctions = TRUE)

#runMCMC
smcMCMCout <- runMCMC(cMCMCsmc,
                      niter = nIterations,
                      nchains = nChains,
                      nburnin = nBurnin,
                      inits = inits,
                      setSeed = TRUE,
                      samples=TRUE,
                      samplesAsCodaMCMC = TRUE,
                      summary = TRUE,
                      WAIC = FALSE)

    #save posterior samples
nimblesmcReducedSamples <- smcMCMCout$samples
```

4.4 Saving all samples

We save all the results from the three fitted reduced SSMs in a list to use in the updating process

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

5 Converting MCMC output for updating process

We load the save 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 three models can be generalised 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"

```

```
$nimblesmcReducedSamples
[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 [**cite nimble manual**] for details on modelValues defnition). To do this, we need to re-define our model is we used JAGS as a nimble model, or use our reduced model from the MCMC with `nimble` and pMCMC with `nimbleMCMC`. 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.

```r
jagsToNimbleCode <- nimbleCode({
  x[1] ~ dnorm(mu0, 1)
  y[1] ~ dnorm(x[1], 1)
  for(i in 2:t){
    x[i] ~ dnorm(x[i-1] * a + b, 1)
    y[i] ~ dnorm(x[i] * c, 1)
  }
  a ~ dunif(0, 1)
  b ~ dnorm(0, 1)
  c ~ dnorm(1,1)
  mu0 ~ dnorm(0, 1)
})


data <- list(
  y = simData$y[-c((iNodePrev+1):50)]
)
# Defining constants
```

```r
constants <- list(
  t = iNodePrev
)
# Initial values
inits <- list(
  a = 0.1,
  b = 0,
  mu0= 0.2,
  c = 1
)
# Define the nimbleModel with the various components defined
jagsToNimbleModel <- nimbleModel(jagsToNimbleCode,
                              data = data,
                              constants = constants,
                              inits = inits,
                              check = FALSE)
```

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

```r
# Define latent states and top-level parameters
latent = "x"
target = c("a", "b", "c", "mu0")

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

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

#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 parameter
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)
```

```r
  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)
```

```
modelValues object with variables: x, a, b, c, mu0.
```

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

```
[1] 1
```

```
# resize the modelValues object created to be equal to (number of iterations - burnin samp
mcmcSampleSize = (nIterations - nBurnin)
resize(mvSamplesEst, mcmcSampleSize)

# check size now
mvSamplesEst$getSize()
```

```
[1] 5000
```

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

#create mvEst for the first chain
for(iter in 1:mcmcSampleSize ){
  for(j in 1:length(names)){
    if(names[j] == latent & length(size[[1]]) > 1){
    mvSamplesEst[[names[j]]][[iter]] <- matrix(mcmcOut[iter, jagsToNimbleModel$expandNodeN
    }else{
      mvSamplesEst[[names[j]]][[iter]] <-  mcmcOut[iter, jagsToNimbleModel$expandNodeNames
    }
  }
}
```

This process of converting MCMC posterior samples to modelValues that cam be used by `nimbleSMC` is written in **updateUtils** function in our `nimbleMCMCSMC` package. It takes as input the reduced model, the MCMC samples, latent state, target vectors, the MCMC sample size, the timeIndes for the **findLatentNodes** function. The code delow show produce the same results as above.

```
    updateVars <- updateUtils(model = jagsToNimbleModel,
                  mcmcOut = mcmcSamplesList$jagsReducedSamples[[1]],
                            latent = latent,
                target = target,
                n.iter = mcmcSampleSize ,
                m = nParFiltRun,
                timeIndex = 1)
```

6 Updating process using nimbleSMC

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 repeated for number of chains times.

Re-defining the nimbleModel

The reduced SSM needs to be updated with the observation from time $t = 46$ 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
)
```

```r
# Initial values
inits <- list(
  a = 0.1,
  b = 0,
  mu0= 0.2,
  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

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

```
===== Monitors =====
thin = 1: a, b, c, mu0, x
===== Samplers =====
(no samplers assigned)
===== Comments =====
```

```r
# 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 = mvSamplesEst,
target = target,
control = pfControl)
```

210

211 Build, compile and run MCMC

```
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 = iNodePrev),
                                          pfNparticles = nParFiltRun,
                                          pfType = pfTypeUpdate,
                                          mvSamplesEst = mvSamplesEst,
```

212

```
                                        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 = mcmcSampleSize,
                            nchains = 1,
                            nburnin = 0,
                            setSeed = TRUE,
                            samples=TRUE,
                            samplesAsCodaMCMC = TRUE,
                            summary = TRUE,
                            WAIC = FALSE)

#check the output
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 three chains.

```
# 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', 'b', 'c', 'mu0'), #top-level parameters
                           additionalPars = "x", #additional parameters to monitor
                           n.iter = mcmcSampleSize, #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 = "lgSSMUpdatedResults.RData")
```

The **spartaNimUpdates** function returns the posterior samples, summary of the posterior samples and the timeRun

```
data("lgssmUpdated") # load results
#str(example1UpdatedModel)
```

6.1 Comparing the various models fit

Our package provides functions to compare the reduced and updated SSM fitted in this document.

6.1.1 Comparing individual model parameters

The *compareModelsIndividualPars* function compares a list of SSM models fit with our package. It returns the efficiency, effective sample size (ESS), time run, and Monte Carlo standard error of the individual models provided in the function. We present the efficiency of the reduced and updated SSM fitted in this document in Table **??**

```
target = c('a', 'b', 'c', 'mu0')
models <- list(example1ReducedModel,
               example1UpdatedModel)
results <- compareModelsIndividualPars(models = models,
                                        n.chains = 2,
                                       nodes = target,
```

```
                                           modelNames = c("reducedModel", "updatedModel"))


# Return efficiency
results$efficiency%>%
  do.call("rbind", .)%>%
  mutate(model = c(rep("reduced Model", length(target)),
                   rep("updated Model", length(target))))%>%
    kbl() %>%
 kable_material(c("striped", "hover"))
```

6.1.2 Comparison plots for selected parameters

The *compareModelsPlots* function also compares a list of SSM models fit with our package. It returns a plot of the efficiency, effective sample size (ESS), time run, and Monte Carlo standard error of the individual models provided in the function. We present the plot of the efficiency of the reduced and updated SSM fitted in this document as shown in Figure **??**.

```
ret <- compareModelsPlots(models = models,
                          modelNames = c("reduced model", "updated Model"),
                   n.chains = 2,
                   nodes = target)
# return efficiency
ret$efficiencyPlot
```

6.1.3 Posterior summaries for models

We also provide a function that returns the posterior mean of parameters of interest and the credible intervals around the estimates.

```
ret <- compareModelsMeanPlots(models = models,
                              modelNames = c("reduced model", "updated Model"),
                              fullModel = stateSpaceModel,
                              nodes = target)


#return confint plot
ret$confintPlot
```

6.1.4 Run baseline 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.

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.