

Example codes for “Backward importance sampling for online estimation of state space models”

Anonymous authors

Contents

1	About the codes	1
2	Comparison for the Sine Model	1
2.1	Installation of the required package	1
2.2	Code for the comparison	2
3	Online estimation in the Sine model	5
3.1	Simulating data	6
3.2	Estimation	7
4	Experiments for the Lotka Volterra Model	8
4.1	Installation of the required package	8
4.2	Smoothing on synthetic data	8
4.3	Experiments on Lynx data	10
5	Stochastic volatility model	13

1 About the codes

- *All the codes presented below are R codes that should work for a R version ≥ 3.4 .*
- The proposed algorithm is mainly embedded in two R packages, one for the first section (that also contains code for the state of the art *Grand PaRIS* algorithm) and one for the second. To read the raw code of the proposed algorithm, the reader should navigate in the `src` directory of the packages after having downloaded them.

2 Comparison for the Sine Model

2.1 Installation of the required package

The required codes are packages into the `GrandParisPackage` package for the R software, available on github. To install this package in R, simply run:

```
devtools::install_github("papayoun/GrandParisPackage")
```

2.2 Code for the comparison

Then, the full comparison is done in the following code:

```
# Cleaning -----

rm(list = ls())

# Packages -----

library(GrandParisPackage) # For the main algorithm
library(parallel) # For parallel computing (do not work on windows)
library(tidyverse) # For data processing
library(tictoc) # For time comparisons
library(gridExtra) # For multiple plots

# Simulating data -----

my_seed <- 333 # For all experiments, the random seed

set.seed(my_seed) # For reproducibility
trueTheta <- pi / 4; trueSigma2 <- 1;

n <- 11; times <- seq(from = 0, to = 5, length = n);
SINEprocess <- SINE_simulate(theta = trueTheta, sigma2 = trueSigma2,
                             x0 = 10, times = times)
observations <- SINEprocess[, "observations"]

# Experiment_function -----

# A function that computes the approximated estimate of  $E[X_0 | Y_{0:n}]$ 
# for different  $\tilde{N}$ 
# It does it 400 times

compare_on_ntilde <- function(n_particle = 100, n_tilde = NULL){
  if(is.null(n_tilde)){
    stop("You must provide ntilde!")
  }
  n_rep <- 400
  AR <- do.call(rbind,
                mclapply(1:n_rep, function(i){
                  tic()
                  res <- GrandParisPackage::E_track(observations = observations,
                                                       ind_tracked = 0,
                                                       observationTimes = times,
                                                       thetaStart = trueTheta,
                                                       particleSize = n_particle,
                                                       backwardSampleSize = n_tilde,
                                                       sigma2Start = trueSigma2,
```

```

nIterations = 1)

my_tictoc <- toc()
dur <- my_tictoc$toc - my_tictoc$tic
names(dur) <- NULL
data.frame(N = n_particle,
           N_tilde = n_tilde, Xhat = res, Time = dur,
           Method = factor("AR",
                           levels = c("AR", "IS")))
},
mc.cores = detectCores() - 1)) # Number of cores for parallelization
IS <- do.call(rbind,
mclapply(1:n_rep, function(i){
  tic()
  res <- GrandParisPackage::E_track_IS(observations = observations,
                                       ind_tracked = 0,
                                       observationTimes = times,
                                       thetaStart = trueTheta,
                                       particleSize = n_particle,
                                       backwardSampleSize = n_tilde,
                                       sigma2Start = trueSigma2,
                                       nIterations = 1)

  my_tictoc <- toc()
  dur <- my_tictoc$toc - my_tictoc$tic
  names(dur) <- NULL
  data.frame(N = n_particle,
             N_tilde = n_tilde, Xhat = res, Time = dur,
             Method = factor("IS",
                             levels = c("AR", "IS")))
},
mc.cores = detectCores() - 1)) # Number of cores for parallelization
return(rbind.data.frame(AR, IS))
}

# Obtaining estimates -----

set.seed(my_seed) # For reproducibility
my_ntildes <- c(2, 5, 10, 20, 30)

# Run only once (experiment) -----

n_tilde_comparison_results <- lapply(my_ntildes, function(n_t)
  compare_on_ntilde(n_particle = 100, n_tilde = n_t)) %>%
  bind_rows()

# Plotting results -----

main_plot <- ggplot(n_tilde_comparison_results,
                   aes(x = factor(N_tilde), fill = Method)) +
  labs(x = expression(tilde(N))) +
  scale_fill_viridis_d(option = "G") +
  theme(legend.position = c(.9, .125))

```

```

p1 <- main_plot +
  geom_boxplot(aes(y = Time)) +
  scale_y_continuous(trans = "log10") +
  labs(y = "Comput. time (seconds)", fill = "")
p2 <- main_plot +
  geom_boxplot(aes(y = Xhat)) +
  labs(y = expression(hat("\U01d53c") ~ "[" ~ X[0] ~ "|" ~ Y[0:n] ~ "]"),
    fill = "")
my_plot <- gridExtra::grid.arrange(p1, p2, nrow = 1)
# 12 * 4 inches

# Controlling N -----

compare_on_n <- function(n_particle = 100, # Number of particles
  alphas = NULL, # Power of N to obtain Ntilde
  # 0.5 and 0.6 in the article
  frac = NULL){ # Fraction of N to obtain Ntilde (0.1 in the article)

  n_tilde_AR <- 2
  levels_IS <- paste0("IS_a_", alphas, "_p_", frac)
  n_rep <- 50
  AR <- do.call(rbind,
    mclapply(1:n_rep, function(i){
      tic()
      res <- GrandParisPackage::E_track(observations = observations,
        ind_tracked = 0,
        observationTimes = times,
        thetaStart = trueTheta,
        particleSize = n_particle,
        backwardSampleSize = n_tilde_AR,
        sigma2Start = trueSigma2,
        nIterations = 1)

      my_tictoc <- toc()
      dur <- my_tictoc$toc - my_tictoc$tic
      data.frame(N = n_particle,
        N_tilde = n_tilde_AR, Xhat = res, Time = dur,
        Method = factor("AR",
          levels = c("AR", levels_IS)))
    }, mc.cores = detectCores() - 2))
  IS <- do.call(rbind.data.frame,
    mapply(function(my_alpha, my_prop){
      n_tilde_IS = ceiling(my_prop * n_particle^my_alpha)
      do.call(rbind.data.frame,
        mclapply(1:n_rep, function(i){
          tic()
          res <- GrandParisPackage::E_track_IS(observations = observations,
            ind_tracked = 0,
            observationTimes = times,
            thetaStart = trueTheta,
            particleSize = n_particle,
            backwardSampleSize = n_tilde_IS,
            sigma2Start = trueSigma2,
            nIterations = 1)

          my_tictoc <- toc()

```

```

        dur <- my_tictoc$toc - my_tictoc$tic
        names(dur) = NULL
        data.frame(N = n_particle,
                   N_tilde = n_tilde_IS, Xhat = res, Time = dur,
                   Method = factor(paste0("IS_a_", my_alpha, "_p_", my_prop),
                                   levels = c("AR", levels_IS)))
      }, mc.cores = detectCores() - 2))
    }, alphas, frac, SIMPLIFY = F))
  return(rbind.data.frame(AR, IS))
}

my_n_particles <- c(50, 100, 200, 500, 1000, 2000)
set.seed(my_seed)
res_npart <- lapply(my_n_particles,
                   function(my_n) compare_on_n(n_particle = my_n,
                                                alphas = c(0.5, 0.6, 1),
                                                frac = c(1, 1, 0.1)))

res_df_npart <- do.call(rbind.data.frame, res_npart)
# save(res_df_npart, file = "res_df_npart.RData")
# Plotting results

my_labels <- expression("AR,"~tilde(N)==2, "IS,"~tilde(N)== N^0.5,
                        "IS,"~tilde(N)== N^0.6, "IS,"~tilde(N)== N/10)
main_plot <- res_df_npart %>%
  mutate(Method = factor(Method, labels = my_labels)) %>%
  ggplot(aes(x = factor(N), fill = Method)) +
  labs(x = expression(N), fill = "") +
  scale_fill_viridis_d(labels = my_labels, option = "G") +
  theme(legend.text.align = 0,
        legend.position = c(.75, .2),
        legend.background = element_blank())

p1 <- main_plot +
  geom_boxplot(aes(y = Time)) +
  scale_y_continuous(trans = "log10") +
  labs(y = "Comput. time (seconds)")
p2 <- main_plot +
  geom_boxplot(aes(y = Xhat)) +
  labs(y = expression(hat("\U1d53c")~"["~X[0]~"|"~Y[0:n]~""]"))
my_plot <- gridExtra::grid.arrange(p1, p2, nrow = 1)

ggplot(res_df_npart, aes(x = N, y = Time)) +
  geom_point() + geom_smooth() +
  facet_wrap(~Method, scales = "free_y") +
  labs(x = expression(tilde(N)),
       y = "Comput. time (seconds)")

```

3 Online estimation in the Sine model

In this section, the online estimation is performed

```
rm(list = ls())
library(GrandParisPackage)
library(tidyverse)
library(parallel)
library(gridExtra)
```

3.1 Simulating data

3.1.1 Simulation parameters

```
# True parameters
trueTheta <- pi/4; trueSigma2 <- 1;
n <- 5000; times <- seq(0, by = 1, length = n)
base_pow <- 0.6
# Function to obtain a sequence of gradient steps
get_grad_steps <- function(gradient_power, cst = 8, n){
  firstStep <- 0.5
  nStart <- 300 # Number
  steps <- c(rep(firstStep, nStart), sapply(cst * firstStep * (1:(n - nStart)) ^ (-gradient_power),
                                             function(x) min(x, firstStep)))
  matrix(steps, ncol = 2, nrow = n)
}
```

3.1.2 Simulation

```
# The following code creates 500 trajectories in a directory "simulated_data"
# that can be created with the following code
# dir.create("simulated_data")
n_traj <- 500
# For windows user, replace by lapply and remove the mc.cores argument below
if(!dir.exists("simulated_data/")){
  dir.create("simulated_data") # Create a folder to stock simulated data
}
# Simulated 500 trajectories and write it in the folder "simulated_data"
# Simulation is done exactly using exact algorithms
mclapply(1:n_traj,
  function(i){
    seed <- 100 + i
    set.seed(seed)
    simulated_POD <- SINE_simulate(theta = trueTheta,
                                   sigma = trueSigma2, x0 = 0,
                                   times = times)
    write.table(simulated_POD, paste0("simulated_data/simul_data_seed", seed, ".txt"),
                col.names = T, row.names = F, sep = ";")
  },
  mc.cores = detectCores() - 1)
```

3.2 Estimation

Code for estimation is only shown for one trajectory. This gives the result of the first part of section 5.3 (the Figure 4)

```
# Get the appropriate trajectory
seed <- 122
set.seed(seed) # For reproducibility
simulated_POD <- read.table(paste0("simulated_data/simul_data_seed", seed, ".txt"),
                           sep = ";", header = TRUE)
observations <- simulated_POD[, "observations"]

# Estimation parameters -----

## One set of observations, several starting points for the algorithm

get_estimation_one_obs_several_start <- function(){
  gradientSteps <- get_grad_steps(0.6, cst = 8, n = n)
  n_start_points <- 50
  n_particles <- 100
  N_tilde <- n_particles / 10
  allRes <- mclapply(1:n_start_points, function(seed){
    set.seed(seed)
    thetaStart <- runif(1, 0, 2 * pi)
    fastTangOR(observations, times, particleSize = n_particles,
              thetaModel = thetaStart, sigma2 = trueSigma2,
              updateOrders = rep(TRUE, n),
              gradientSteps = gradientSteps,
              all = FALSE, estimateTheta = TRUE, estimateSigma2 = FALSE,
              randomWalkParam = 1, backwardSampleSize = N_tilde, IS = TRUE)
  },
  mc.cores = detectCores() - 1)
  thetaEst <- sapply(allRes, function(x) x$Estimates[,1]) %>% (2*pi)
  return(thetaEst)
}

res_one_obs_several_start <- get_estimation_one_obs_several_start()

res_several_obs_one_start <- mclapply(1:length(dir("simulated_data/")), function(i){
  seed <- 100 + i
  set.seed(seed)
  simulated_POD <- read.table(paste0("simulated_data/simul_data_seed", seed, ".txt"),
                           sep = ";", header = T)
  observations <- simulated_POD[, "observations"]
  thetaStart <- trueTheta
  Res <- fastTangOR(observations, times,
                  thetaModel = thetaStart, sigma2 = trueSigma2, particleSize = 500,
                  updateOrders = rep(TRUE, length(observations)),
                  gradientSteps = get_grad_steps(0.6, cst = 8,
                                                  n = length(observations)),
                  all = TRUE, estimateSigma2 = FALSE, randomWalkParam = 1)

  Res
```

```
},
mc.cores = detectCores() - 1)
```

4 Experiments for the Lotka Volterra Model

4.1 Installation of the required package

The required codes are packages into the LotkaVoltR package for the R software, available on github.

To install this package in R, simply run:

```
devtools::install_github("papayoun/LotkaVoltR")
```

4.2 Smoothing on synthetic data

4.2.1 Simulating data

```
# Cleaning

rm(list = ls()) # Cleaning environment

# Librairies

library(LotkaVoltR) # Dedicated library
library(tidyverse) # For data processing

# Dynamics parameters

a1 <- c(12, 0.05, 1) # Prey parameters
a2 <- c(2, 0.2, 0.1) # Predator parameters
Gamma <- matrix(c(0.5, 0.1, 0.1, 0.2), nrow = 2) # Diffusion parameters
mu0 <- c(50, 20) # Mean of the initial distribution
Sigma0 <- diag(1, 2) # Variance of the initial distribution

# Observation process parameters

Sigma_obs <- matrix(c(0.01, 0.005, 0.005, 0.01), ncol = 2) # Observation noise
q_values <- c(0.2, 0.3) # Known q values

# Model creation

# Creation of a Partially Observed Lotka Volterra model
POLV_model <- POLV_create(a1 = a1, a2 = a2, gam = Gamma, mu0 = mu0,
                          sigma0 = Sigma0, cov = Sigma_obs, qs = q_values)

# Synthetic data simulation

# Simulation times for the Euler scheme

simulation_times <- seq(from = 0, by = 1e-6, # Simulation time step, small!!)
```



```

length.out = 3*10^6 + 1)

# If the simulation must be done at small time steps, the output can be thinned
# Here we keep 301 points

selection <- seq(1, length(simulation_times), length.out = 301)
set.seed(333)
simulated_process <- POLV_simulate(POLV_model,
                                   times = simulation_times,
                                   selection = selection)

# Extracting observations for smoothing

observation_times <- simulation_times[selection]
observations <- simulated_process[, c("Y1", "Y2")]

```

4.2.2 Performing smoothing

```

parameters_list <- list(a1 = a1, a2 = a2, mu0 = mu0,
                        sigma0 = Sigma0,
                        RWC = diag(0.005, 2), # Parameter for the particle filter
                        qs = q_values, cov = Sigma_obs,
                        wD = 1, w0 = 1,
                        gam = Gamma)
particle_filter <- PF_create(parameters_list, t(observations),
                             observation_times, 2e2,
                             n_euler_skel = 30)

# Performing the smoothing
# This creates the particles and their (filtering) weights, together
# with computing the wanted  $E[X_k | Y_{0:n}]$  for all  $k$ 

smoothing_exp <- particle_filter$runSmoothing()

```

4.2.3 Plotting results

```

# Creating labels for the plot
my_levels <- c(obs = 'Y[t]',
               est = 'hat("\U1d53c")~["~X[t]~"|"~Y[0:n]~"]"',
               true = 'X[t]')

smoothing_mean <- smoothing_exp %>%
  as_tibble() %>%
  rename(X1 = V1, X2 = V2) %>%
  mutate(method = my_levels["est"])
observed_values <- simulated_process %>%
  as.data.frame() %>%
  select(Y1, Y2) %>%
  mutate(Y1 = Y1 / q_values[1], # Putting back in the actual state space

```

```

      Y2 = Y2 / q_values[2]) %>%
  rename(X1 = Y1, X2 = Y2) %>%
  mutate(method = my_levels["obs"])
true_values <- simulated_process %>%
  as.data.frame() %>%
  select(X1, X2) %>%
  mutate(method = my_levels["true"])

concatened_results <- bind_rows(observed_values,
                                true_values,
                                smoothing_mean) %>%
  mutate(method = factor(method, levels = my_levels)) # Pour l'ordre
ggplot(concatened_results) +
  aes(x = X1, y = X2, color = method, linetype = method) +
  geom_point() +
  geom_path() +
  theme(legend.position = "none") +
  facet_wrap(~method, labeller = label_parsed) +
  labs(x = "Number of preys", y = "Number of predators") +
  theme(strip.text = element_text(size = 24))

```

4.3 Experiments on Lynx data

4.3.1 Data set

```

rm(list = ls()) # Cleaning environment
hares_lynx_data <- read.table("supplementary_data_hares_lynx.txt",
                             skip = 4, header = TRUE, sep = "\t")

```

4.3.2 Performing EM for estimation

Initial parameter

```

# Dynamics parameters
## Initial values inspired from https://gist.github.com/mages/1f0f0d5bbe50af81cc19

a1 <- c(0.7, 1e-3, 3e-2) # Prey parameters
a2 <- c(0.6, 2e-2, 1e-3) # Predator parameters
Gamma <- diag(.2, 2) # Diffusion parameters
mu0 <- c(30, 4) # Mean of the initial distribution
Sigma0 <- diag(1, 2) # Variance of the initial distribution

# Observation process parameters

Sigma_obs <- matrix(c(0.01, 0.00, 0.00, 0.01), ncol = 2) # Observation noise
q_values <- c(1, 1) # Known q values
initial_param <- list(a1 = a1, a2 = a2, mu0 = mu0,
                     sigma0 = Sigma0, RWC = diag(0.005, 2),
                     qs = q_values, cov = Sigma_obs,
                     wD = 1, w0 = 1,

```

```

        gam = Gamma)

# Model creation

```

EM functions: A generalized EM is performed. At each step, candidates are generated using a home made (not optimized!) evolution strategy, by sampling around the current parameter through a Gaussian distribution. Each parameter lives in a constrained space. Each new offspring is generated in the unconstrained \mathbb{R} space before being put back to the constrained space. This is mainly done with the logistic function.

Then, a usual EM approach is performed. To ensure good fit, the EM is performed from three different starting points.

```

library(LotkaVotrR) # Dedicated library

# Function generation candidat -----

generate_candidate_list <- function(par_list, # Current parameter
                                   standard_dev_list, # Standard Deviation for
                                   # the future offspring generation
                                   up_lim_list){ # Limits of the constrained space

  out <- par_list
  # Generating a1, in [0, 1] * [0, 0.05] * [0, 0.05]
  sigmoid <- function(x){
    1 / (1 + exp(-x))
  }
  logit <- function(x){
    log(x / (1 - x))
  }
  out$a1 <- (par_list$a1 / up_lim_list$a1) %>% # Back to [0, 1]
    logit() %>% # Back to the real world
    rnorm(n = 3, sd = standard_dev_list$a1) %>% # Moving a bit
    sigmoid() %>% # Retour dans [0,1]
    {. * up_lim_list$a1} # Back to the constrained space
  out$a2 <- (par_list$a2 / up_lim_list$a2) %>% # Back to [0, 1]
    logit() %>% # Back to the real world
    rnorm(n = 3, sd = standard_dev_list$a2) %>% # Moving a bit
    sigmoid() %>% # Retour dans [0,1]
    {. * up_lim_list$a2} # Back to [0, 1] * [0, 0.1] * [0, 0.1]
  diag(out$gam) <- (diag(par_list$gam) / up_lim_list$gam) %>% # Back to [0, 1]
    logit() %>% # Back to the real world
    rnorm(n = 2, sd = standard_dev_list$gam) %>% # Moving a bit
    sigmoid() %>% # Back to [0,1]
    {. * up_lim_list$gam}
  diag(out$cov) <- (diag(par_list$cov) / up_lim_list$cov) %>% # Back to [0, 1]
    logit() %>% # Back to the real world
    rnorm(n = 2, sd = standard_dev_list$cov) %>% # Moving a bit
    sigmoid() %>% # Back to [0,1]
    {. * up_lim_list$cov}
  return(out)
}

EM_function <- function(obs, obs_times, initial_param, initial_sd, up_lims,
                        n_cands, n_iter,

```

```

        seed,
        name = NULL){
param_0 <- initial_param
out <- list(param_0)
set.seed(seed)
final_E_steps <- matrix(NA, nrow = n_iter, ncol = n_cands + 1)
for(i in 1:n_iter){
  print(paste("Iteration", i))
  par_list <- c(purrr::rerun(n_cands,
                           generate_candidate_list(param_0,
                                                    initial_sd,
                                                    up_lims)),
               list(param_0))
  E_step_evals <- get_E_step(obs_ = t(obs),
                           obsTimes_ = obs_times,
                           myParams = param_0,
                           testedParams = par_list,
                           n_part = 200, n_dens_samp = 100)
  initial_sd <- purrr::map(initial_sd,
                           function(x) 0.9 * x) # Reducing the exploration
  param_0 <- par_list[[which.max(E_step_evals)]]
  out <- c(out, list(param_0))
  final_E_steps[i, ] <- E_step_evals
}
return(list(out, final_E_steps))
}

observations <- as.matrix(hares_lynx_data[, c("Hares", "Lynx")])
observation_times <- hares_lynx_data$Year

initial_param <- list(a1 = a1, a2 = a2, mu0 = mu0,
                     sigma0 = Sigma0, RWC = diag(0.005, 2),
                     qs = q_values, cov = Sigma_obs,
                     wD = 1, w0 = 1,
                     gam = Gamma)
sd_list <- list(a1 = c(1, 1, 1),
               a2 = c(1, 1, 1),
               cov = 0.5,
               gam = 0.5)
upper_lims_list <- list(a1 = c(1, 0.01, 0.05),
                       a2 = c(1, 0.05, 0.01),
                       cov = 0.5,
                       gam = 0.5)
library(parallel) # For parallel computing
results <- mclapply(1:30, function(my_seed){
  EM_function(obs = observations,
              obs_times = observation_times,
              initial_param = initial_param,
              initial_sd = sd_list,
              up_lims = upper_lims_list,
              n_cands = 10,
              n_iter = 30, seed = my_seed,
              name = "lynx")},

```

```
mc.cores = detectCores() - 1)
```

The results are then used to perform the smoothing as in previous section.

5 Stochastic volatility model

Smoothing functions for the stochastic volatility model are available in the file `utils_SV_PF_functions.R`. The user can change the `smoothing` argument in the code below to test other smoothing methods (such as PoorMan or FFBSi).

```
# Clean environment -----

rm(list = ls())

# Loading data -----

library(tidyverse)
library(parallel)
library(stochvolTMB) # To get the data

# Data

Ys <- get(data(spy, package = "stochvolTMB"))

# Loading particle filter functions -----

source("utils_SV_PF_functions.R")

# Performing PF -----

set.seed(123)

# Checking an E_step -----

# Different starting points, inspired by the vignette of the stochvolTMB package
params_design <- expand.grid(beta_ = seq(1e-3, 5e-2, length.out = 5),
                             phi_ = c(0.9, 0.95, 0.99),
                             sigma_ = c(0.1, 0.15, 0.2))
foo <- function(beta_, phi_, sigma_, n_EM_steps = 30){
  param0 <- list(beta = beta_, phi = phi_, sigma = sigma_)
  results <- data.frame(beta = beta_, phi = phi_, sigma = sigma_,
                        iteration = 0)
  for(i in 1:n_EM_steps){
    PF <- get_particle_filter(Ys = Ys,
                             params = param0,
                             method = "bootstrap",
                             N = 400,
                             smoothing = "BIS",
                             IS_method = "IS",
                             get_N_tilde = function(N) .5 * N)
    param0 <- get_exact_M_step(length(Ys), PF$E_statistics)
```

```

results <- bind_rows(results,
  data.frame(beta = param0$beta,
    phi = param0$phi,
    sigma = param0$sigma,
    iteration = i))
}
last_PF <- get_particle_filter(Ys = Ys,
  params = param0,
  method = "bootstrap",
  N = 1e3,
  smoothing = "none")
list(res_df = results, likelihood = last_PF$loglik)
}
all_results <- mcmapply(foo, params_design$beta_,
  params_design$phi_,
  params_design$sigma_,
  mc.cores = detectCores() - 1,
  SIMPLIFY = FALSE)
# Get the best final param
best_param <- all_results[[map_dbl(all_results, "likelihood") %>% which.max()]]$res_df %>%
  slice(n()) %>%
  as.list()

# Final smoother, using FFBSi
get_particle_filter(Ys = Ys,
  params = best_param,
  method = "bootstrap",
  N = 100,
  smoothing = "FFBSi")

```