SMC

Ant Stephenson, Shannon Williams

5/25/2021

Introduction

This document provides a (very) brief overview of Sequential Monte Carlo (SMC) methods and Stochastic Volatility (SV) models alongside an implementation in both `R` and `Rcpp` to demonstrate state (and parameter) estimation in this context. We start by introducing SMC, followed by a similar introduction of SV models. This leads us on to demonstrate our implementation(s) of a bootstrap particle filter for state estimation, before finally an implementation of the SMC^2 algorithm, which simultaneously estimates the parameters and the states.

Sequential Monte Carlo

Sequential Monte Carlo methods refer to a set of simulation-based filtering algorithms that enable us to compute posterior distributions for Bayesian state-space models (among others). SMC methods include *particle filters* in their various guises and extend Bayesian filtering capabilities beyond the widely known *Kalman* filter (and its own extensions) to nonlinear, non-Gaussian models.

Stochastic Volatility Models

Generally, Stochastic Volatility (SV) models refer to financial models of log-returns ($Y_t = \log(p_t/p_{t-1})$) where the volatility is taken to be a stochastic process. This is as opposed to deterministic models such as (G)ARCH ([Generalised] auto-regressive conditional heteroscedastic).

The model we shall use here is the following

$$Y_t|X_t = x_t \sim \mathcal{N}(0, \exp(x_t))$$

where $\{X_t\}$ is an auto-regressive process following

$$X_t = \mu + \rho(X_{t-1} - \mu) + U_t$$

with $U_t \sim \mathcal{N}(0, \sigma^2)$.

Data Generation

We define a function to generate data according to the SV model:

```r
update_Xt <- function(mu, rho, Ut, Xs) {
  Xt <- rho * (Xs - mu) + Ut + mu
  return(Xt)
}

#' @export generate_SV_data
generate_SV_data  <- function(mu, rho, sigma, tmax) {
  U <- rnorm(tmax, 0, sigma)
  X0 <- rnorm(1, 0, 1)
  Xt <- c(X0)
  for (t in 2:(tmax+1)) {
    Xt <- c(Xt, update_Xt(mu, rho, U[t-1], Xt[t-1]))
  }
  return(Xt)
}
```

Implementation

To model the latent variables in the SV model we implement a bootstrap particle filter in `R`. To aid further generalisation, we define an API using an OOP approach, by defining a framework based around the Feynman-Kac formalism and implementing a specific class for the SV model. The particle filter then relies only on the Feynman-Kac structure and could in principle take any object implementing that API (though here we only implement the specific SV class).

```r
#' Systematic resampling algorithm
#'
#' Samples a single uniform random variable U and then assigns values U(n) = (n - 1 + U)/N.
#' Used to generate a vector of integers to index the particles.
#' @name systematic_resampling
#' @export systematic_resampling
systematic_resampling <- function(W) {
  N <- length(W)
  v <- cumsum(W)
  s <- runif(1, 0, 1/N)
  A <- rep(0, N)
  m <- 0
  for (n in 1:N) {
    while (v[n] > s) {
      m <- m + 1
      s <- s + 1/N
    }
    A[n] <- ifelse(m == 0, 1, m)
  }
  return(A)
}

#' Bootstrap filter
#'
#' Implements a bootstrap particle filter.
#' Takes an object of class Bootstrap_SV_C as argument, with number of particles
#' N and number of time steps to run.
#' @name bootstrap_filter_rcpp
#' @export bootstrap_filter_rcpp
#' @field fk_model Bootstrap_SV_C object
#' @field N number of particles
#' @field tmax number of steps
bootstrap_filter <- function(fk_model, N, tmax,
                             resampling = systematic_resampling,
                             essmin_fn = function(N) N/2) {
  # compute threshold
  essmin <- essmin_fn(N)
  # initialise simulated values of X
  x <- matrix(NA, nrow = (tmax + 1), ncol = N)
  # sample N times from the prior
  x[1, ] <- fk_model$sample_m0(N)
  # initialise vector of ess
  ess <- c()

  # initialise weights
  w <- matrix(NA, nrow = (tmax + 1), ncol = N) # w_t
  W <- matrix(NA, nrow = (tmax + 1), ncol = N) # W_t
  hw <- matrix(NA, nrow = tmax, ncol = N) # hat{w}_t
  w[1, ] <- fk_model$logG(1, x[1, ])
  W[1, ] <- exp(w[1, ]) / sum(exp(w[1, ]))

  # initialise mean and sd output
  mx <- c()
  sdx <- c()
  mx[1] <- sum(W[1, ] * x[1, ])
  sdx[1] <- sum((x[1, ] - mx[1])^2) / (N-1)

  # initialise ancestor variables
  A <- matrix(NA, nrow = tmax, ncol = N)
  # resampling times
  r <- c()

  for (t in 2:(tmax+1)) {
    # compute ess
    ess[t-1] <- eff_particle_no(W[t-1, ])
    # resampling step
    if (ess[t-1] < essmin) {
      r <- c(r, t-1)
      A[t-1, ] <- resampling(W[t-1, ])
      hw[t-1, ] <- rep(0, N)
    } else {
      A[t-1, ] <- 1:N
      hw[t-1, ] <- w[t-1, ]
    }
    # draw X_t from transition kernel
    x[t, ] <- fk_model$sample_m(x[t-1, A[t-1, ]])
    # update weights
    w[t, ] <- hw[t-1, ] + fk_model$logG(t, x[t, A[t-1, ]])
    W[t, ] <- exp(w[t, ]) / sum(exp(w[t, ]))
    # update mean and sd output
    mx[t] <- sum(W[t, ] * x[t, ])
    sdx[t] <- sqrt(sum((x[t, ] - mx[t])^2) / (N-1))
  }
  return(list(A = A, x = x, hw = hw, w = w, W = W,
              mx = mx, sdx = sdx, r = r, ess = ess))
}

bootstrap_onestep <- function(fk_model, N,
                              resampling = systematic_resampling,
                              essmin_fn = function(N) N/2) {
  # sample N times from the prior of X
  x <- matrix(fk_model$sample_m0(N), ncol = N, nrow = 1)
```

```r
  # initialise weights
  w <- matrix(fk_model$logG(1, x[1, ]), ncol = N, nrow = 1) # w_t
  W <- matrix(exp(w[1, ]) / sum(exp(w[1, ])), ncol = N, nrow = 1) # W_t
  if (eff_particle_no(W) < essmin_fn(N)) {
    A <- matrix(resampling(W), ncol = N, nrow = 1)
  } else {
    A <- matrix(1:N, ncol = N, nrow = 1)
  }
  # draw X_t from transition kernel
  x <- matrix(fk_model$sample_m(x[, A[1, ]]), ncol = N, nrow = 1)
  return(list(A = A, x = x))
}

#' Bootstrap class for Stochastic Volatility (SV) model
#'
#' Initialised with a data vector of observations (Yt), an estimate mean and
#' standard deviation parameter and rho controls the autocorrelation.
#'
#' The object defines methods to sample from the initial distribution M0 and
#' the subsequent transition kernels Mt and calculate the log probability from
#' the potential function Gt.
#' @name Bootstrap_SV_C
#' @export Bootstrap_SV_C
#' @exportClass Bootstrap_SV_C
#' @field data vector of observations
#' @field mu estimate of mean of latent var (float)
#' @field sigma estimate of sd of latent
#' @field rho autocorrelation parameter
require(methods)
Bootstrap_SV <- setRefClass("Bootstrap_SV",
                             fields = list(
                               data = "matrix",
                               mu = "numeric",
                               sigma = "numeric",
                               rho = "numeric",
                               tmax = "numeric",
                               sigma0 = "numeric"),
                             methods = list(
                               initialize = function(data, mu = 0, sigma = 1, rho = 0.95) {
                                 .self$data = data
                                 .self$tmax = length(data)
                                 .self$mu = mu
                                 .self$sigma = sigma
                                 .self$rho = rho
                                 .self$sigma0 = sigma / sqrt(1 - rho^2)
                               },
                               sample_m0 = function(N) {
                                 rnorm(N, mean = .self$mu, sd = .self$sigma0)
                               },
                               sample_m = function(xp) {
                                 rnorm(length(xp),
                                       mean = .self$mu + .self$rho * (xp - .self$mu),
                                       sd = .self$sigma)
```

```
                                },
                                logG = function(t, x) {
                                  dnorm(.self$data[t], mean = 0,
                                        sd = sqrt(exp(x)), log = TRUE)
                                }
                              ))
```

and additionally in `C++` making use of the `Rcpp` package in order to make simulations more computationally manageable. We go on to test that a) the results match (excluding variations due to unset random seeds in C++) and b) that the C++ implementation is indeed faster than in `R`. We aim to replicate the API defined in the `R` code above in `Rcpp` for ease of use.

```
#include <Rcpp.h>
#include "utils.h"
#include "particles.h"
using namespace Rcpp;

//' Effective Particle Number
//'
//' Uses the weight vector input to compute an estimate of effective particle number for use in particl
//' @name eff_particle_no
//' @export eff_particle_no
//'
//' @field w vector of weights
//[[Rcpp::export]]
double eff_particle_no(
    const NumericVector w
)
{
  double ess = 1.0/sum(w * w);
  return ess;
}

//' Systematic resampling algorithm
//'
//' Samples a single uniform random variable U and then assigns values U(n) = (n - 1 + U)/N.
//' Used to generate a vector of integers to index the particles.
//' @name systematic_resampling_rcpp
//' @export systematic_resampling_rcpp
// [[Rcpp::export(name = "systematic_resampling")]]
IntegerVector systematic_resampling(const NumericVector W) {
  int N = W.length();
  NumericVector v = cumsum(W);
  float inc = 1.0/(float)N;
  float s = R::runif(0, 1) * inc;
  IntegerVector A(N);
  int m = 0;
  for (int n=0; n<N; n++) {
    while(v[n] > s + inc) {
      m += 1;
      s += inc;
    }
    A[n] = (m == 0) ? 1 : m;
  }
  return A;
}

//' Bootstrap class for Stochastic Volatility (SV) model
//'
//' Initialised with a data vector of observations (Yt), an estimate mean and
//' standard deviation parameter and rho controls the autocorrelation.
//'
//' The object defines methods to sample from the initial distribution M0 and
//' the subsequent transition kernels Mt and calculate the log probability from
//' the potential function Gt.
//' @name Bootstrap_SV_C
//' @export Bootstrap_SV_C
//' @exportClass Bootstrap_SV_C
//' @field data vector of observations
//' @field mu estimate of mean of latent var (float)
//' @field sigma estimate of sd of latent
//' @field rho autocorrelation parameter
Bootstrap_SV_C::Bootstrap_SV_C(NumericVector data, float mu, float sigma, float rho)
{
  this->_data = data;
  this->_mu = mu;
  this->_sigma = sigma;
  this->_rho = rho;
  this->tmax = data.length();
  this->sigma0 = sigma / sqrt(1.0 - rho * rho);
}

NumericVector Bootstrap_SV_C::sample_m0(int N)
{
  return rnorm(N, this->_mu, this->sigma0);
}

NumericVector Bootstrap_SV_C::sample_m(NumericVector xp)
{
  int N = xp.length();
  NumericVector sample(N);

  for (int i=0; i<N; i++) {
    sample[i] = R::rnorm(this->_mu + this->_rho * (xp[i] - this->_mu), this->_sigma);
  }
  return sample;
}

NumericVector Bootstrap_SV_C::logG(int t, NumericVector x)
{
  int N = x.length();
  NumericVector logg(N);

  for (int i=0; i<N; i++) {
    logg[i] = R::dnorm(this->_data[t], 0.0, sqrt(exp(x[i])), true);
  }
  return logg;
```

```
}


float essmin_fn(int N) {
  return (float)N/2.0;
}

//' Bootstrap filter
//'
//' Implements a bootstrap particle filter.
//' Takes an object of class Bootstrap_SV_C as argument, with number of particles
//' N and number of time steps to run.
//' @name bootstrap_filter_rcpp
//' @export bootstrap_filter_rcpp
//' @field fk_model Bootstrap_SV_C object
//' @field N number of particles
//' @field tmax number of steps
List bootstrap_filter_rcpp(Bootstrap_SV_C fk_model, int N, int tmax) {//, float(*f)(int) = [](int N) {r
  //float essmin = (*f)(N);
  float essmin = essmin_fn(N);

  // initialise simulated values of X
  NumericMatrix x(tmax+1, N);
  // sample N times from the prior
  x(0, _) = fk_model.sample_m0(N);

  // initialise mean and sd output
  NumericVector mx(tmax+1);
  NumericVector sdx(tmax+1);

  // initialise ess storage
  IntegerVector ess(tmax);

  // initialise weights
  NumericVector w(N);
  NumericMatrix W(tmax+1, N);
  NumericVector hw(N);

  w = fk_model.logG(1, x(0, _));
  W(0, _) = exp(w) / sum(exp(w));

  // initialise ancestor variables
  IntegerMatrix A(tmax, N);

  // resampling times
  IntegerVector r;

  for (int t=1; t<=tmax; t++) {
    ess(t-1) = eff_particle_no(W(t-1, _));
    if (ess(t-1) < essmin) {
      r.push_back(t-1);
      A(t-1, _) = systematic_resampling(W(t-1, _));
      hw = 0 * hw;
      hw = 0*hw;
    }
    else {
      for (int i=0; i<N; i++) {
        A(t-1, i) = i + 1;
      }
      hw = w;
    }
    IntegerVector s = A(t-1, _);
    s = s - 1;

    // draw X_t from transition kernel
    NumericVector xp = ncindex(x, t-1, s);
    x(t, _) = fk_model.sample_m(xp);

    // update weights
    NumericVector xt = ncindex(x, t, s);
    w = hw + fk_model.logG(t, xt);
    W(t, _) = exp(w) / sum(exp(w));

    // update mean and sd output
    mx(t) = sum(W(t, _) * x(t, _));
    sdx(t) = sqrt(sum(pow(x(t, _) - mx(t), 2.0) / (float)(N-1)));
  }

  List output = List::create(_["A"] = A, _["x"] = x, _["hw"] = hw, _["W"] = W,
                             _["mx"] = mx, _["sdx"] = sdx, _["r"] = r, _["ess"] = ess);

  return output;
}

//' One-step bootstrap filter
//'
//' Implements a bootstrap particle filter with T=1.
//' Takes an object of class Bootstrap_SV_C as argument, with number of particles N.
//' @name bootstrap_onestep_rcpp
//' @export bootstrap_onestep_rcpp
//' @field fk_model Bootstrap_SV_C object
//' @field N number of particles
List bootstrap_onestep_rcpp(Bootstrap_SV_C fk_model, int N) {
  float essmin = essmin_fn(N);

  // initialise simulated values of X
  NumericVector x(N);
  // sample N times from the prior
  x = fk_model.sample_m0(N);

  // initialise weights
  NumericVector w(N);
  NumericVector W(N);

  w = exp(fk_model.logG(1, x));
  W = w / sum(w);
  // initialise ancestor variables
  IntegerVector A(N);

  if (eff_particle_no(W) < essmin) {
    A = systematic_resampling(W);
  }
  else {
    for (int i=0; i<N; i++) {
      A(i) = i + 1;
    }
  }

  // convert A to index
  IntegerVector s = clone(A);
  s = s - 1;

  // draw X_1 from transition kernel
  NumericVector xp(N);
  xp = x[s];
  x = fk_model.sample_m(xp);

  List output = List::create(_["A"] = A, _["x"] = x);
  return output;
}


// Expose class and function to R using RcppModules.

RCPP_EXPOSED_CLASS(Bootstrap_SV_C);

RCPP_MODULE(particles) {

  Rcpp::class_<Bootstrap_SV_C>("Bootstrap_SV_C")

  .constructor<Rcpp::NumericVector, float, float, float>()

  .field("_data", &Bootstrap_SV_C::_data)
  .field("tmax", &Bootstrap_SV_C::tmax)
  .field("_mu", &Bootstrap_SV_C::_mu)
  .field("_sigma", &Bootstrap_SV_C::_sigma)
  .field("_rho", &Bootstrap_SV_C::_rho)
  .field("sigma0", &Bootstrap_SV_C::sigma0)

  .method("sample_m0", &Bootstrap_SV_C::sample_m0)
  .method("sample_m", &Bootstrap_SV_C::sample_m)
  .method("logG", &Bootstrap_SV_C::logG)
  ;

  function("bootstrap_filter_rcpp", &bootstrap_filter_rcpp);
  function("bootstrap_onestep_rcpp", &bootstrap_onestep_rcpp);
}

/*** R
```

```
# set.seed(1)
#
# tmax <- 100
# mu <- -1
# rho <- 0.95
# sigma <- 0.15
#
# N <- 1000
#
# Xt <- generate_SV_data(mu, rho, sigma, tmax)
# Yt <- as.matrix(rnorm(tmax, mean = 0, sd = sqrt(exp(Xt))))
#
# boot_sv <- new(Bootstrap_SV_C, Yt, mu, sigma, rho)
#
# output <- bootstrap_filter_rcpp(boot_sv, N, tmax)
*/
```

First generate some data according to the SV model.

```
library(smc)
```

```
##
## Attaching package: 'smc'

## The following objects are masked _by_ '.GlobalEnv':
##
##     bootstrap_filter, bootstrap_onestep, Bootstrap_SV,
##     generate_SV_data, systematic_resampling, update_Xt
```

```
library(Rcpp)

set.seed(1)

tmax <- 500
mu <- -1
rho <- 0.95
sigma <- 0.15
N <- 5000

Xt <- generate_SV_data(mu, rho, sigma, tmax)
Yt <- as.matrix(rnorm(tmax, mean = 0, sd = sqrt(exp(Xt))))
```

Run the `R` particle filter implementation, by first instantiating an SV object with given parameters and then running this through the bootstrap PF.

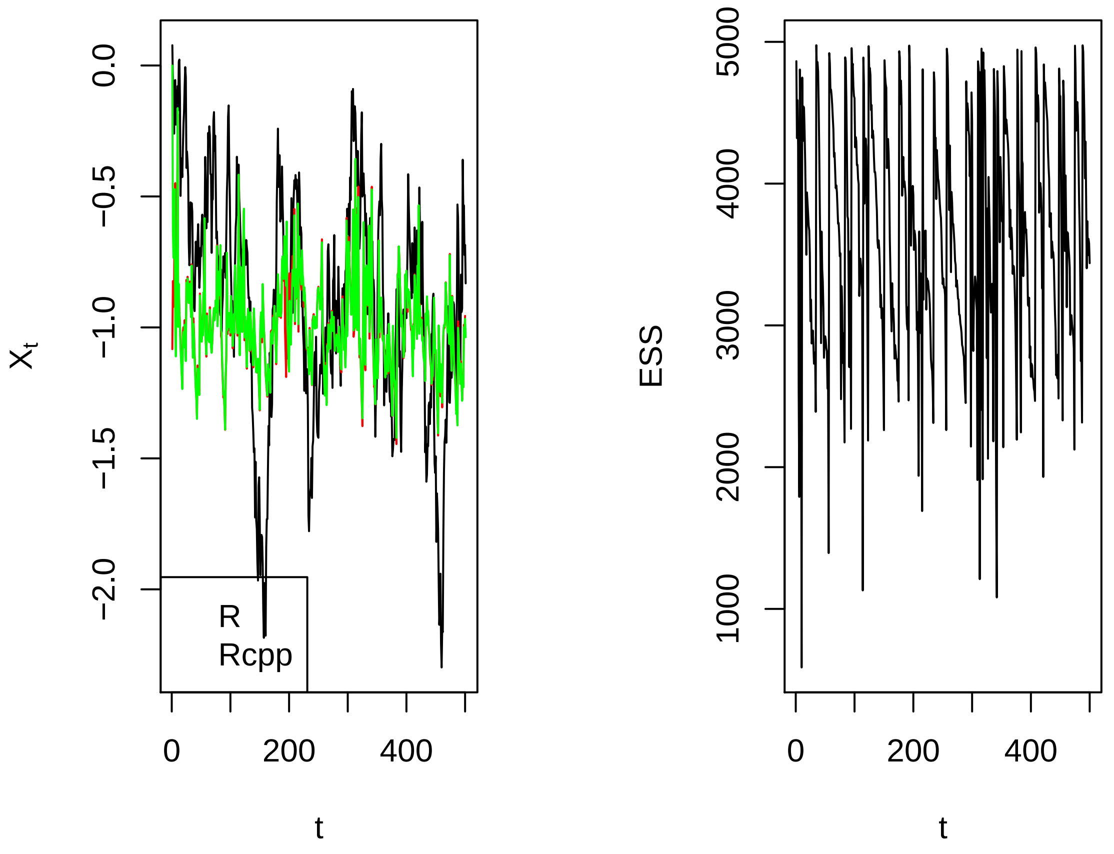
```
boot_sv <- Bootstrap_SV$new(data = Yt, mu = -1, sigma = 0.15, rho = 0.95)

output <- bootstrap_filter(boot_sv, N, tmax)
```

For the `Rcpp` version we must load the associated package exposed to `R`, initialise the `Rcpp` exposed class and then run the particle filter.

```
Bootstrap_SV_C <- Bootstrap_SV_C
boot_sv_rcpp <- new(Bootstrap_SV_C, Yt, -1, 0.15, 0.95)
output2 <- bootstrap_filter_rcpp(boot_sv_rcpp, N ,tmax)
```

Plot the mean outputs from the R and Rcpp versions over the true latent variable. We see that the `Rcpp` version (in green) is more or less identical to the (red) `R` version. The difference we attribute to random seeds.

```
library(latex2exp)
# pdf("docs/figs/state_estimation_pf.pdf", width=8, height=3)
par(mar=c(4,4,1,4))
par(mfrow = c(1,2))
plot(Xt, type = "l", ylab=TeX("$X_t$"), main="", xlab="t")
lines(output$mx, col = "red")
lines(output2$mx, col="green")
legend("bottomleft", legend=c("R", "Rcpp"))
plot(1:tmax, output$ess, type = "l", ylab="ESS", xlab="t")
```



```
# dev.off()

# Delete the outputs to avoid running out of memory
rm(output)
rm(output2)
gc()
```

```
##           used (Mb) gc trigger  (Mb) limit (Mb) max used  (Mb)
## Ncells  714121 38.2    1437370  76.8         NA  1437370  76.8
## Vcells 1349168 10.3   20966623 160.0      16384 26206109 200.0
```

We now verify that the `Rcpp` version is faster than the `R` implementation using the `microbenchmark` package:

```
library(microbenchmark)

run_filter <- function(filter_fn, ...) {
  out <- filter_fn(...)
  rm(out)
  gc()
  return(NA)
}
funcs <- c(call("run_filter", bootstrap_filter, boot_sv, N, tmax), call("run_filter", bootstrap_filter_r

bench_res <- microbenchmark(list=setNames(funcs, c("R","Rcpp")),
                times=5L)

print(bench_res, signif=3)
```

```
## Unit: milliseconds
##   expr  min   lq mean median   uq  max neval
##      R 1330 1330 1370   1350 1390 1440     5
##   Rcpp  545  556  602    558  594  756     5
```

SMC^2

The particle filter implementation above relies on knowing the parameters σ^2, ρ, μ *a priori*. Of course, this is not likely in any realistic scenario, so we must be able to estimate these parameters as well. Here we choose to implement the SMC^2 algorithm to simultaneously estimate the parameters alongside state estimation with the particle filter.

We run the SMC^2 algorithm to estimate the parameters μ and σ, using a Gaussian random walk proposal for both μ and σ. Since parameter estimation means we have an effective burn-in for state estimation, we choose to take the final parameters from the SMC^2 results and use them in an additional bootstrap filtering step.

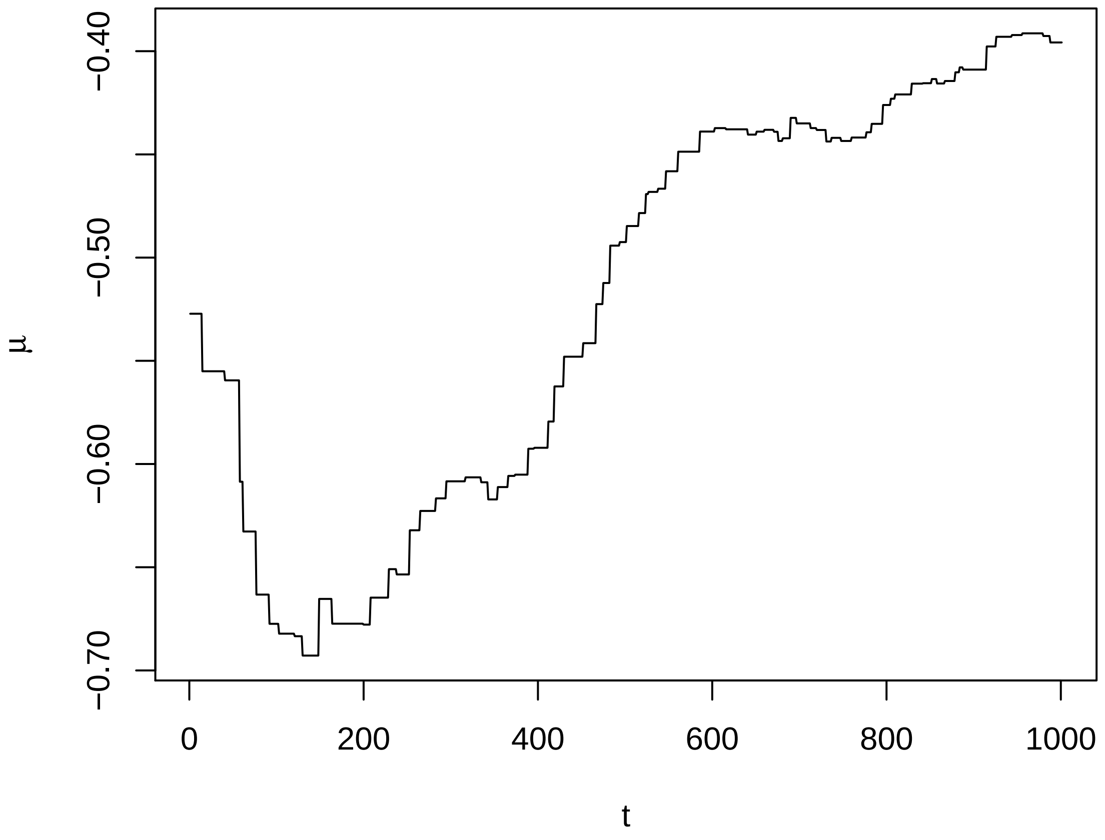
We start by estimating only the μ parameter, before attempting to extend to σ. We choose to avoid attempting to estimate ρ due to computational difficulties associated with the tight prior needed to constrain the values to something close to (but less than or equal to) 1. This seems reasonable for the kind of SV setting we are considering due to the generally accepted assumption that it be so constrained that an assumed value close to 1 will suffice.

```
tmax <- 1000
mu <- -0.99
rho <- 0.95
sigma <- 0.15
Xt <- generate_SV_data(mu, rho, sigma, tmax)
Yt <- as.matrix(rnorm(tmax+1, mean = 0, sd = sqrt(exp(Xt))))
Nx <- 500
Nt <- 200
mu_prior <- -0.5
sd_prior <- 0.3
sd_prop <- 0.5
smc_results <- smc_squared_rcpp(Yt, Nx, Nt, sigma, rho,
                                mu_prior = mu_prior, sd_prior = sd_prior, sd_prop =sd_prop)
thetas <- smc_results$thetas
Wm <- smc_results$Wm
rm(smc_results)
```

Before we run the bootstrap filter again, first analyse the output from SMC^2. We start by plotting the convergence of the parameters:

```
# pdf("docs/figs/param_estimation_smc2.pdf", width=8, height=3)
par(mar=c(4,4,1,4))
par(mfrow = c(1,1))
```

```
plot(rowMeans(thetas), ylab=TeX("$\\mu$"), type="l", xlab="t")
lines(rep(mu, tmax+1), col="red")
```



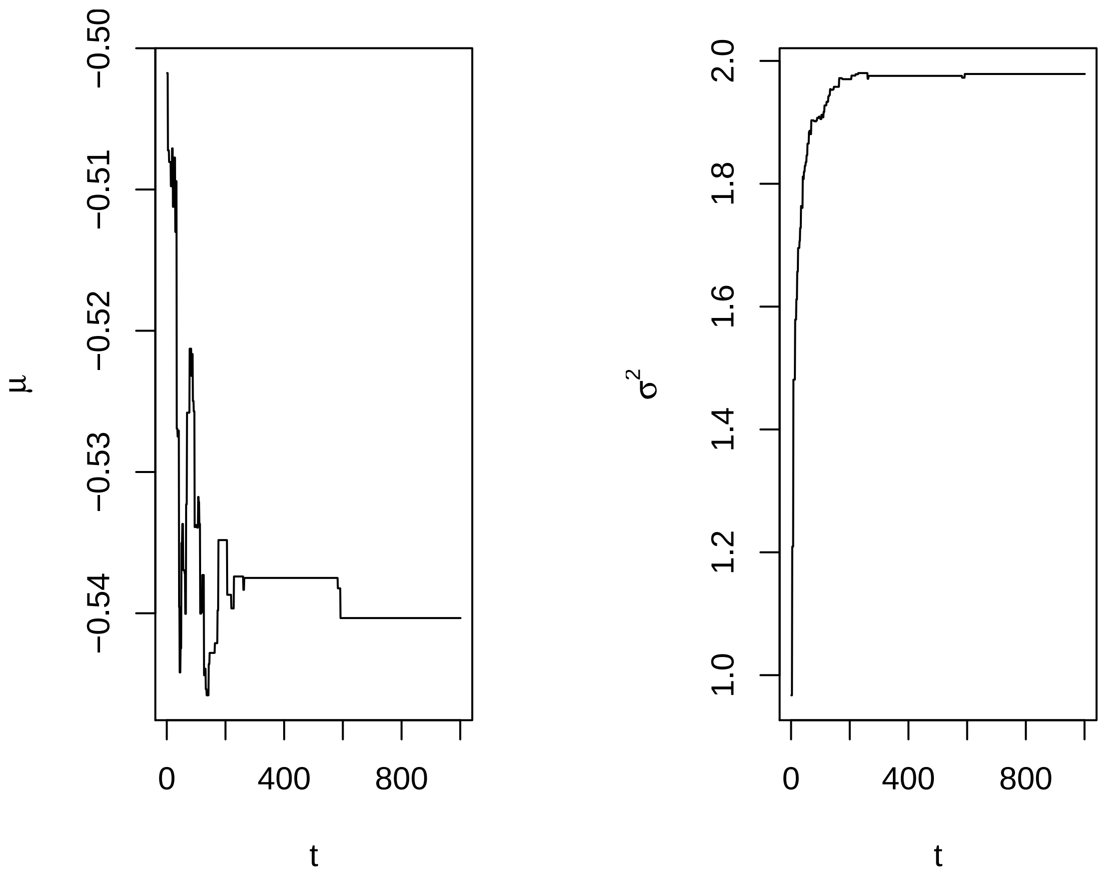
```
# dev.off()
```

We now try to extend the SMC^2 implementation to σ.

```
tmax <- 1000
mu <- -1
rho <- 0.95
sigma <- 0.15
Xt <- generate_SV_data(mu, rho, sigma, tmax)
Yt <- as.matrix(rnorm(tmax+1, mean = 0, sd = sqrt(exp(Xt))))
Nx <- 500
Nt <- 200

mu_prior = c(-0.5, 0.2)
sigma_prior <- c(2, 2)
sd_prop <- c(0.5, 1.0)

smc_results2 <- smc_squared_rcpp2(Yt, Nx, Nt, sigma, rho,
                           mu_prior = mu_prior, sigma_prior = sigma_prior, sd_prop = sd_prop)
thetas2 <- smc_results2$thetas
Wm2 <- smc_results2$Wm
rm(smc_results2)
```

Again we plot the convergence of the parameters, as we did for the single parameter version.

```
# pdf("docs/figs/param_estimation_smc22.pdf", width=8, height=3)
par(mar=c(4,4,1,4))
par(mfrow = c(1,2))
plot(rowMeans(thetas2[,,1]), ylab=TeX("$\\mu$"), type="l",xlab="t")
lines(rep(mu, tmax+1), col="red")
plot(rowMeans(thetas2[,,2]), ylab=TeX("$\\sigma^2$"), type="l",xlab="t")
lines(rep(sigma, tmax+1), col="red")
```



```
# dev.off()
```

We can see that the algorithm does a very poor job of estimating both μ and σ^2 together, so we opt to ignore this implementation when considering the final filtering step and use only the single parameter version

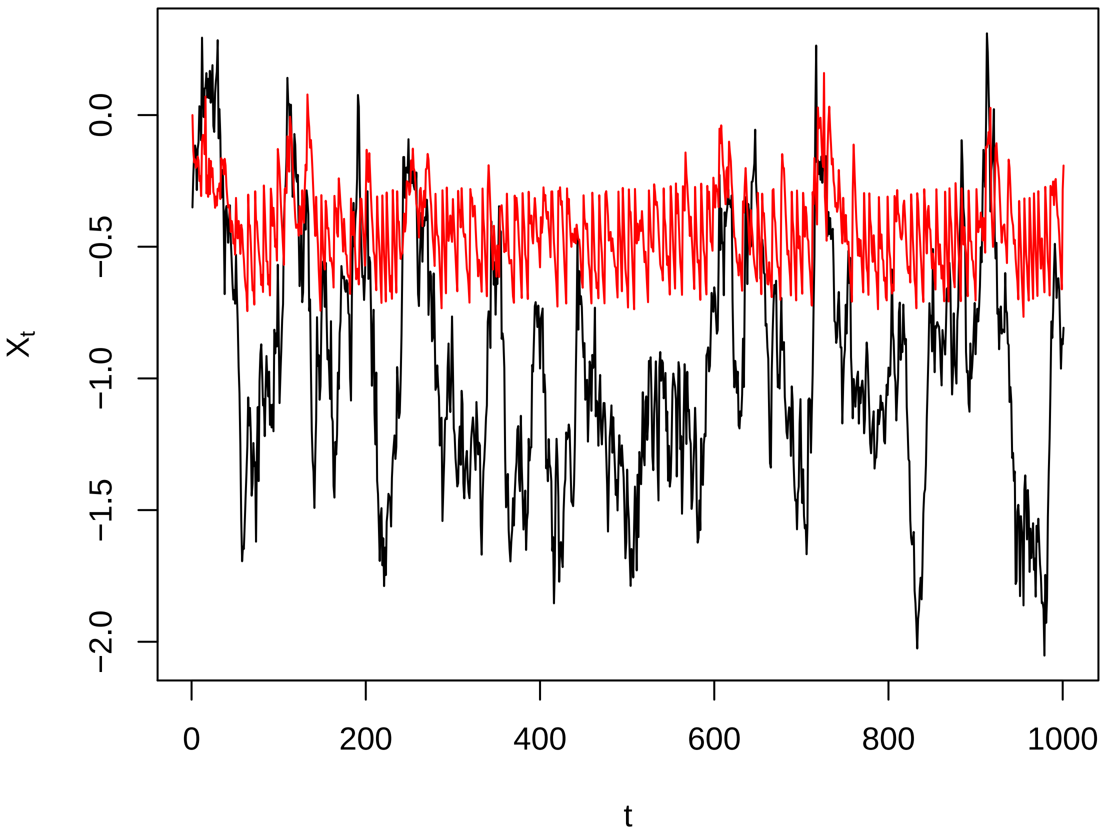
Now to run the bootstrap filter, we use the population of thetas at the final step to get the estimate for the parameters μ and σ to use in the final state estimation step.

```
mut = sum(Wm[tmax+1,] * thetas[tmax+1,])/sum(Wm[tmax+1,])
mut2 = sum(Wm2[tmax+1,] * thetas2[tmax+1,,1])/sum(Wm2[tmax+1,])
sigma_t2 = sum(Wm2[tmax+1,] * thetas2[tmax+1,,2])/sum(Wm2[tmax+1,])
```

Plot the output from the final bootstrap particle filter, using just μ estimated using SMCˆ2.

```
boot_sv_smc2 <- new(Bootstrap_SV_C, Yt, mut, 0.15, 0.95)
output_smc2 <- bootstrap_filter_rcpp(boot_sv_smc2, N ,tmax)

# pdf("docs/figs/state_estimation_smc2.pdf", width=8, height=3)
par(mar=c(4,4,1,4))
par(mfrow = c(1,1))
```

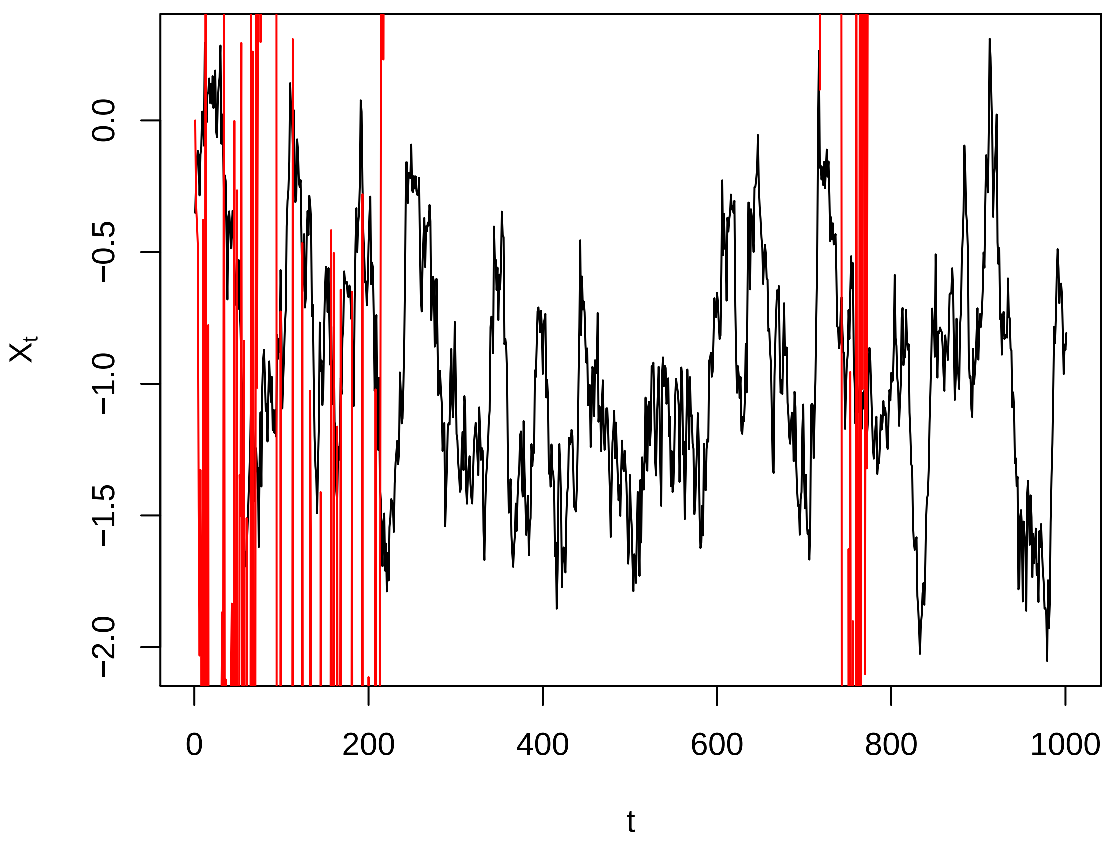
```
plot(Xt, type = "l", ylab=TeX("$X_t$"),xlab="t")
lines(output_smc2$mx, col = "red")
```



```
# dev.off()
```

Plot the output from the final bootstrap particle filter, using both parameters estimated using SMC^2.

```
boot_sv_smc2 <- new(Bootstrap_SV_C, Yt, mut2, sigma_t2, 0.95)
output_smc2 <- bootstrap_filter_rcpp(boot_sv_smc2, N ,tmax)

# pdf("docs/figs/state_estimation_smc22.pdf", width=8, height=3)
par(mar=c(4,4,1,4))
par(mfrow = c(1,1))
plot(Xt, type = "l", ylab=TeX("$X_t$"),xlab="t")
lines(output_smc2$mx, col = "red")
```



```
# dev.off()
```