SMC2017: Exercise set I

Niharika

June 15, 2018

This document provides solution for the SMC2017: Exercise set I given at http://www.it.uu.se/research/systems_and_control/education/2017/smc/homework/SMC2017_exercises1.pdf.

I.1 Importance sampling

(a) Sample from target $\pi(x) = U[0, 4]$ with proposal density $q(x)$ as standard normal. Is this a valid importance sample? Answer is yes, because for a valid importance sampler, as $q(x) > 0, forall x$ and $\pi(x) > 0, for x \in [0, 4]$.

(b) Implement the suggested importance sampler with N = 10,000.

```
targetFunc =  function(x, value=1)
{
  if(x < 0 || x > 4)
  {
    return(0)
  }
  return(value)
}

exactSampling = function(mu, sigma, iter = 10000)
{
  imp_weight = rep(0, iter)

  x <- rnorm(iter, mu , sigma)

  for(i in 1:iter){
    imp_weight[i] = targetFunc(x[i], value = 0.25) / dnorm(x[i], mean =mu , sd=sigma)
  }
  return(list(x=x, imp_weight=imp_weight))
}

importanceSampling = function(mu , sigma, iter = 10000)
{
  imp_weight = rep(0, iter)

  x <- rnorm(iter, mu , sigma)

  for(i in 1:iter){
    imp_weight[i] = targetFunc(x[i], value = 1) / dnorm(x[i], mean =mu , sd=sigma)
  }
  if(sum(imp_weight) == 0)
  {
    stop("Too few samples")
  }
  normalized_weight <- imp_weight/sum(imp_weight)
```

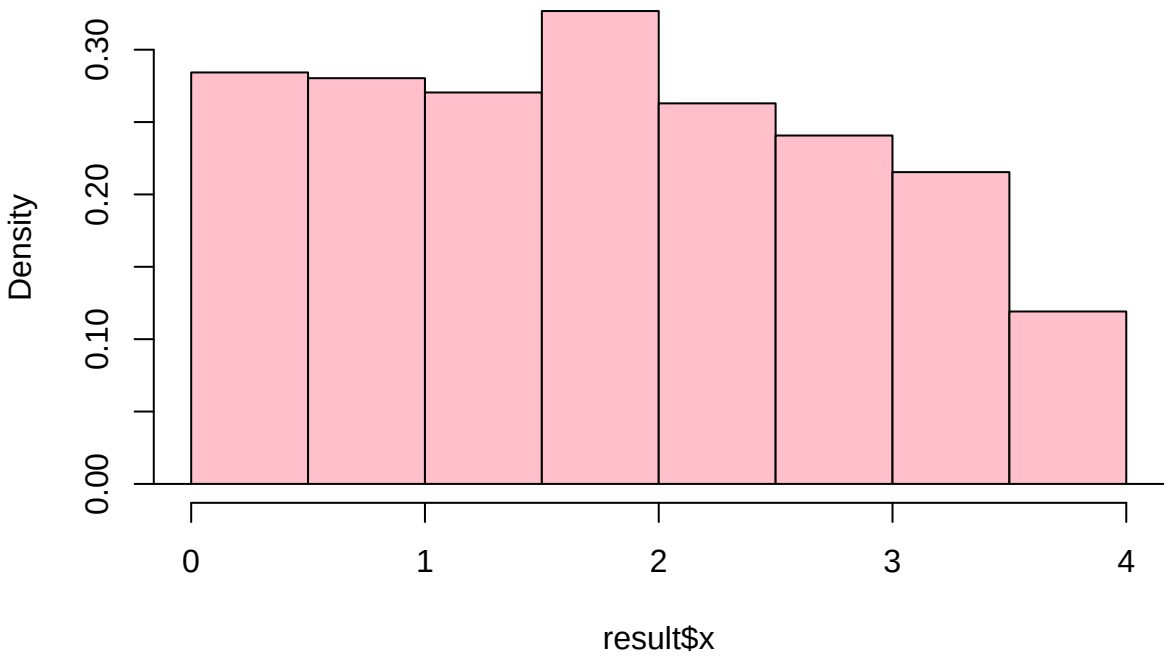
```
  return(list(x=x, imp_weight = imp_weight, normalized_weight=normalized_weight))
}
```

Plot the results of importance sampling.

```
result = importanceSampling(mu = 2, sigma=2, 50000)
wtd.hist(result$x, weight = result$normalized_weight, xlim = range(0,4),
         probability = TRUE, breaks = 50, col="pink")
```

Plot the results of exact sampling. We notice that, when the proposal does not covers the target distribution properly, then the resulting histogram will be distorted image of the target distribution.

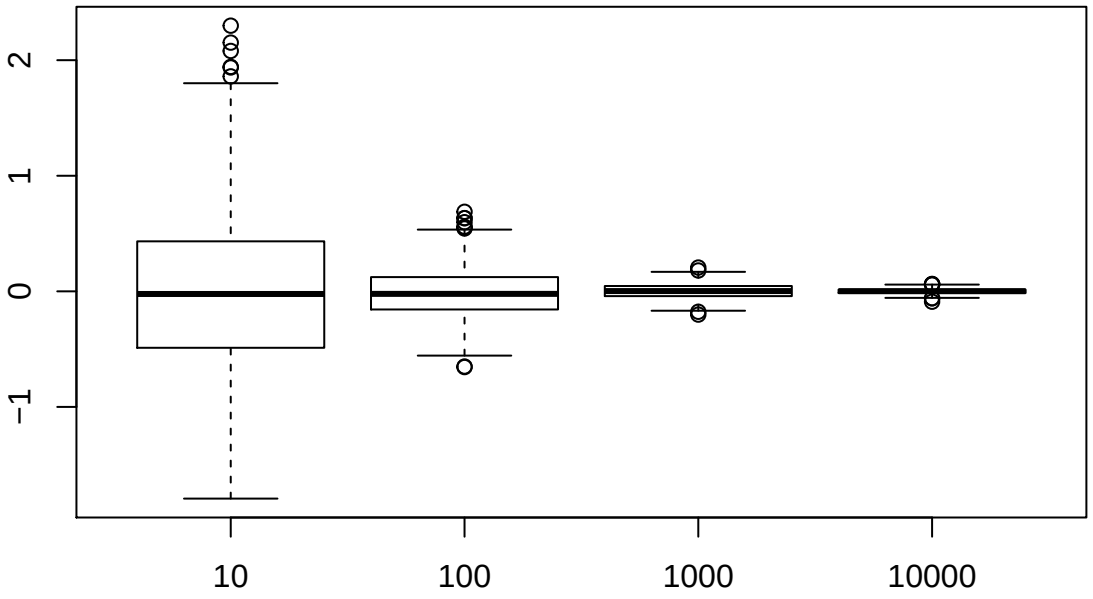
```
result = exactSampling(mu = -3, sigma=2, 10000)
wtd.hist(result$x, weight = result$imp_weight, xlim = range(0,4),
         probability = TRUE, breaks = 50, col="pink")
```



(c) Checking for bias, with various sample sizes.

The mean values of $\pi(x) = U[0, 4]$ is 2. Let us check the biase using simulation.

```
N = c(10, 100, 1000, 10000)
sampleX = matrix(0, 1000, length(N))
for (j in 1:length(N)) {
  for (i in 1:1000) {
    result = exactSampling(mu = 2, sigma=2, iter = N[j])
    sampleX[i,j] = mean(result$x * result$imp_weight)-2
  }
}

colnames(sampleX) = as.character(N)
boxplot(sampleX)
```



Boxplot plot shows that, variance decreases with sample size, however it always gives unbiased estimate.

(d) Estimate for normalizing constant is basically average of simulated weights.

$$\begin{aligned} Z &= \int \tilde{\pi}(x)dx \\ &= \int \frac{\tilde{\pi}(x)}{q(x)} q(x)dx \\ &= \int w(x)q(x)dx \end{aligned}$$

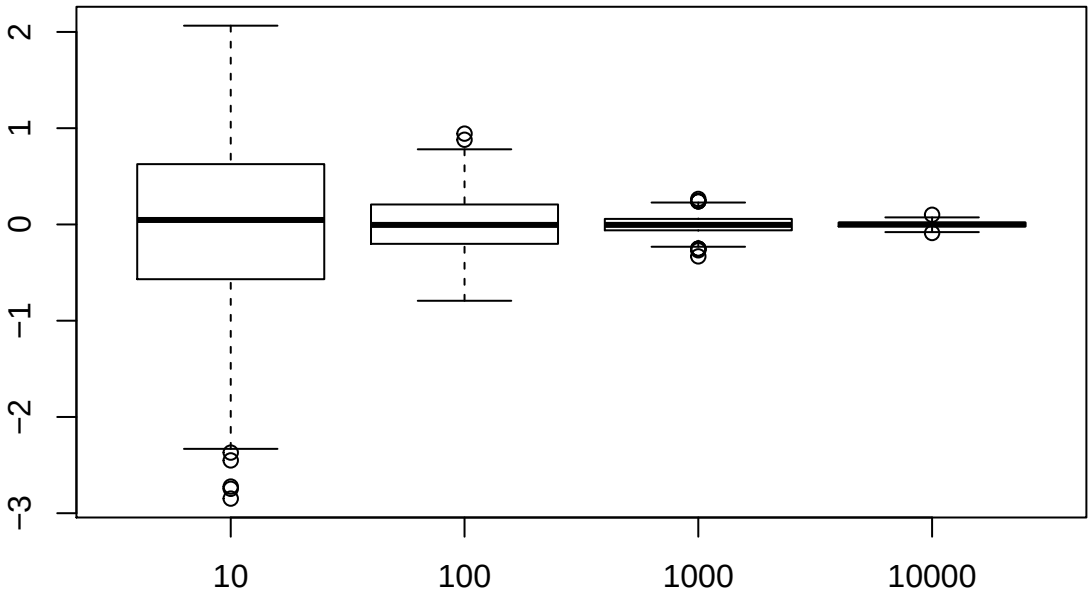
with Monte Carlo estimate, it becomes

$$Z \approx \frac{1}{N} \sum w(x_i)$$

(e) Value of the normalizing constant Z is 4.

Let us check the biase using simulation.

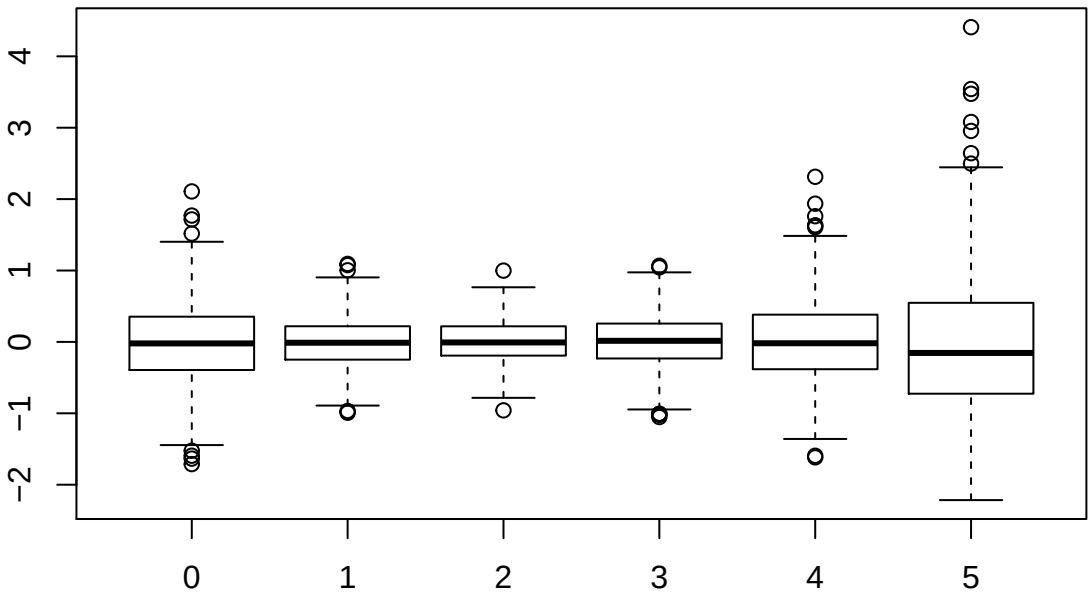
```
N = c(10, 100, 1000, 10000)
sampleX = matrix(0, 1000, length(N))
for (j in 1:length(N)) {
  for (i in 1:1000) {
    result = importanceSampling(mu = 2, sigma=2, iter = N[j])
    sampleX[i,j] = mean(result$imp_weight)-4
  }
}

colnames(sampleX) = as.character(N)
boxplot(sampleX)
```



Boxplot plot shows that, variance decreases with sample size, however it always gives unbiased estimate of Z.

To check the influence of the proposal mean, we keep its standard deviation fixed and move the mean value of the proposal .

```
mu = c(0:5)
sampleX = matrix(0, 1000, length(mu))

for (j in 1:length(mu)) {
  for (i in 1:1000) {
    result = importanceSampling(mu = mu[j], sigma=2, 100)
    sampleX[i,j] = mean(result$imp_weight)-4
  }
}

colnames(sampleX) = as.character(mu)
boxplot(sampleX)
```

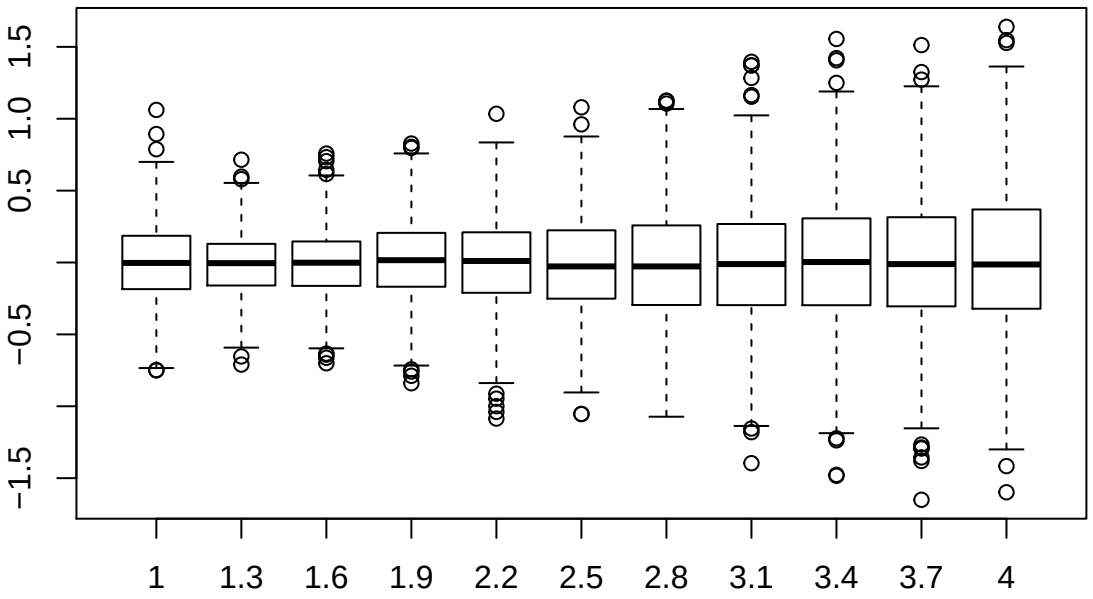


Moving the mean value of the proposal away from the midpoint of the uniform distribution, increases the variance.

To check the influence of the proposal standard deviation, now we keep its mean fixed and move the standard

deviation of the proposal .

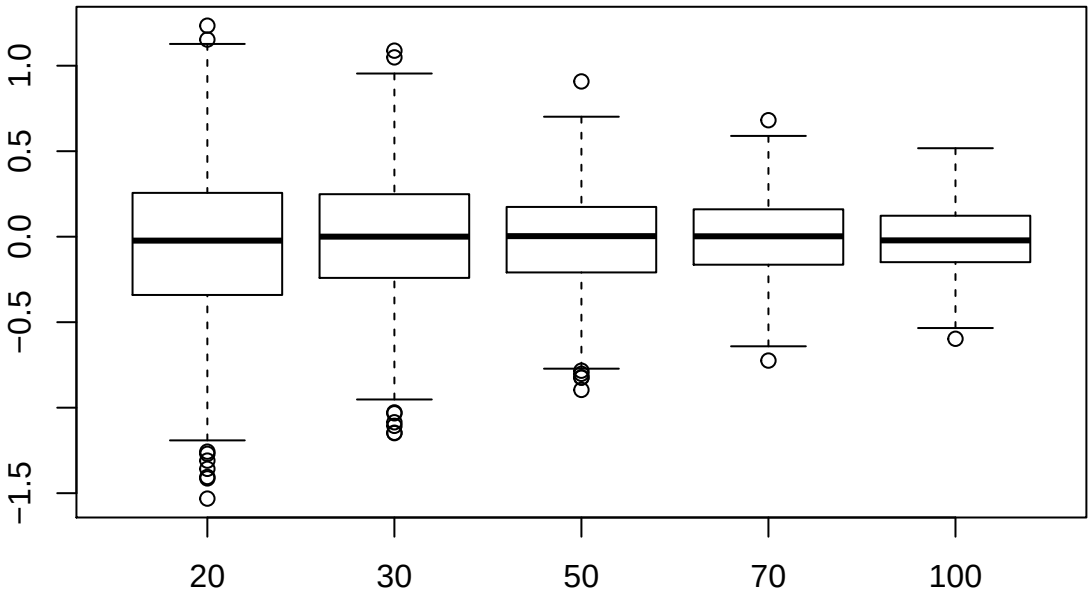
```
sigma = seq(1, 4, length.out = 11)
sampleX = matrix(0, 1000, length(sigma))

for (j in 1:length(sigma)) {
  for (i in 1:1000) {
    result = importanceSampling(mu = 2, sigma=sigma[j], 100)
    sampleX[i,j] = mean(result$imp_weight)-4
  }
}

colnames(sampleX) = as.character(sigma)
boxplot(sampleX)
```



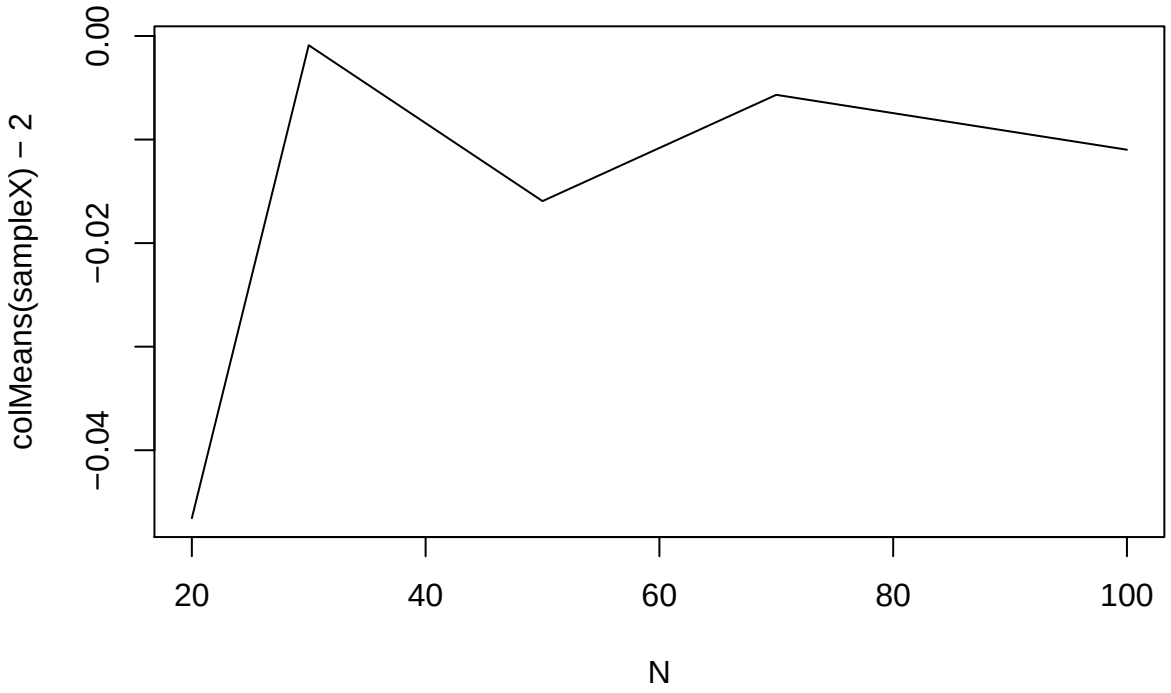
We notice that variance decreases when standard deviation is increased from 1 to 1.3, then it starts increasing with the stansard deviation.

f) Bias of estimator of expected value of the target distribution in case of self-normalized importance sampling

```
N = c(20, 30, 50, 70, 100)
sampleX = matrix(0, 1000, length(N))
for (j in 1:length(N)) {
  for (i in 1:1000) {
    result = importanceSampling(mu = 0, sigma=3, iter = N[j])
    sampleX[i,j] = sum(result$x * result$normalized_weight)
  }
}

colnames(sampleX) = as.character(N)
boxplot(sampleX-2)
```




```
# Plot bias of the estimator for the expected value
plot(N, colMeans(sampleX)-2, type="l")
```

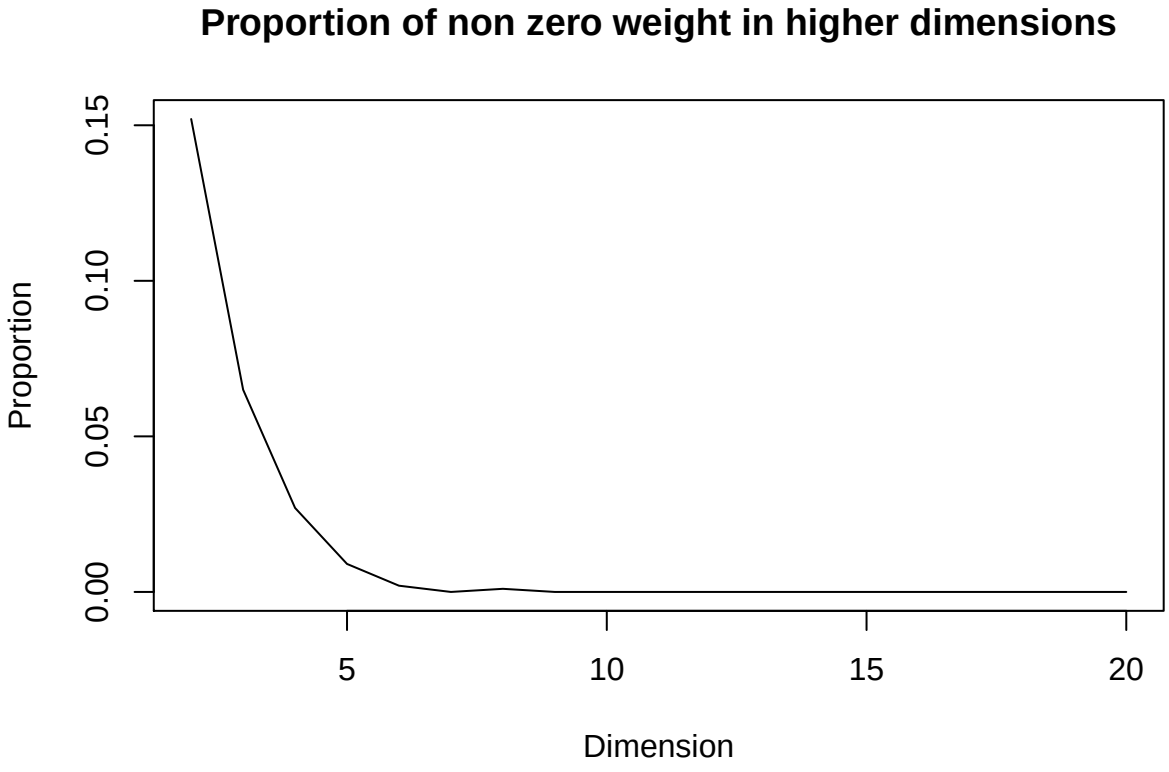


I.2 Importance sampling in higher dimensions.

With target density as $\pi(x) = U_D[-0.5, 0.5]$, and the proposal density as $q(x) = N_D(0, I)$.

```
targetFunc =  function(y)
{
  if(any(y < -0.5))
    return(0)

  if(any(y > 0.5))
    return(0)

  return(1)
}
```

```r
impSampling = function(iter = 1000, d = 2)
{
  weights <- rep(0, iter)
  m <- rep(0, d) #mean vector
  Sigma <- sigma2 * diag(d)
  x <- mvrnorm(iter, m , Sigma)

  for(i in 1:iter){
    weights[i] = targetFunc(x[i,]) / dmvnorm(x[i,], m , Sigma)
  }

  return(list(x=x, weights=weights))
}

sigma2 <- 1
propNonZeroWeight <- rep(0, 19) #proportion of non-zero weight
for(k in 2:20)
{
  result <- impSampling(1000, k)
  w <- result$weights
  propNonZeroWeight[k-1] <- length(w[w>0])/length(w)
}

plot(c(2:20), propNonZeroWeight, type = "l", xlab = "Dimension",
     ylab = "Proportion", main = "Proportion of non zero weight in higher dimensions")
```



The above plot shows that the proportion of non zero weights decreases exponentially with the dimension.

I.3 An important numerical aspect

For the following experiment consider D = 1000.

(a) Simulate 10 samples from the target density $\pi(x) = N_D(0, I)$, with the proposal density as $q(x) = N_D(0, 4 * I)$. Then compute the weights and the normalized weights for them.

```
densityNorm =  function(y, sigma)
{
  z = dnorm(y, mean = 0, sd = sigma)
  return(prod(z))
}

impSamplingNorm = function(iter = 1000, d = 2)
{
  imp_weight = rep(0, iter)
  m <- rep(0, d) #mean vector
  Sigma <- sigma2 * diag(d)
  x <- rmvnorm(iter, m , Sigma)
  pi_x <- rep(0.0, iter)
  q_x <- rep(0.0, iter)

  for(i in 1:iter){
    pi_x[i] <- densityNorm(x[i,],  sigma = 1)
    q_x[i] <- densityNorm(x[i,],  sigma = 2)
    imp_weight[i] <-  pi_x[i]/ q_x[i]
  }
  sum_weight <- sum(imp_weight)
  norm_weight <- imp_weight/sum_weight #normalize weights

  return(list(x=x, imp_weight=imp_weight, norm_weight = norm_weight,
         pi_x = pi_x, q_x = q_x))
}

sigma2 <- 4
result <- impSamplingNorm(10, 1000)
result$imp_weight
```

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

```
result$norm_weight
```

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

```
result$pi_x
```

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

```
result$q_x
```

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

From the above experiment we found that all $\pi(x_i)$ and $q(x)$ are zero though their individual component density of standard normal pdf is non zero.

Therefor all the weights and normalized weights are NaN due to divide by zero.

Importance sampling using log-normal pdf to avoid divide by zero (NaN)

```
impSamplingLogNorm = function(iter = 1000, d = 2)
{
```

```
  imp_weight = rep(0, iter)
  m <- rep(0, d) #mean vector
  Sigma <- sigma2 * diag(d)
  x <- rmvnorm(iter, m , Sigma)
  pi_x <- rep(0.0, iter)
  q_x <- rep(0.0, iter)

  for(i in 1:iter){
    pi_x[i] <- dmvnorm(x[i,],  m, sigma = diag(d), log = TRUE)
    q_x[i] <- dmvnorm(x[i,],  m, sigma = Sigma, log = TRUE)
    imp_weight[i] <-  exp(pi_x[i] - q_x[i])
  }
  sum_weight <- sum(imp_weight)
  norm_weight <- imp_weight/sum_weight

  return(list(x=x, imp_weight=imp_weight, norm_weight = norm_weight,
         pi_x = pi_x, q_x = q_x))
}

sigma2 <- 4
result <- impSamplingLogNorm(10, 1000)
result$imp_weight
```

```
##   [1]  0.000000e+00 4.917559e-301  0.000000e+00  0.000000e+00  0.000000e+00
##   [6] 5.458190e-319  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
```

```
result$norm_weight
```

```
##   [1] 0.000000e+00 1.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
##   [6] 1.109939e-18 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
```

```
result$pi_x
```

```
##   [1] -2898.737 -2765.115 -3001.006 -3095.696 -3046.878 -2820.238 -3013.157
##   [8] -2906.659 -2929.333 -2916.550
```

```
result$q_x
```

```
##   [1] -2107.035 -2073.630 -2132.603 -2156.275 -2144.071 -2087.411 -2135.640
##   [8] -2109.016 -2114.684 -2111.489
```

From the above experiment, we notice that we have completly avoided the divide by zero problem, but still non-zero weights are reported as zero because log(weight) are very small and exp(log(weight)) is smaller that the system can represent. Therefor we get zero weights and zero normalized weights.

The trick of computing weights

```
impSamplingLogNormWithWeightTrick = function(iter = 1000, d = 2)
{
  imp_weight = rep(0, iter)
  m <- rep(0, d) #mean vector
  Sigma <- sigma2 * diag(d)
  x <- rmvnorm(iter, m , Sigma)
  pi_x <- rep(0.0, iter)
  q_x <- rep(0.0, iter)
```

```
  for(i in 1:iter){
    pi_x[i] <- dmvnorm(x[i,],  m, sigma = diag(d), log = TRUE)
    q_x[i] <- dmvnorm(x[i,],  m, sigma = Sigma, log = TRUE)
    imp_weight[i] <-  pi_x[i] - q_x[i]
  }
  max_weight <- max(imp_weight)
  imp_weight <- exp(imp_weight - max_weight)
  norm_weight <- imp_weight / sum(imp_weight)
  return(list(x=x, imp_weight=imp_weight, norm_weight = norm_weight))
}

sigma2 <- 4
result <- impSamplingLogNormWithWeightTrick(10, 1000)
result$norm_weight
```

```
##  [1]  3.925134e-76  1.000000e+00  1.845632e-26  1.986717e-40  7.432498e-35
##  [6]  9.939044e-79 6.955130e-140  7.752548e-60  1.046191e-72  1.120967e-91
```

From the above experiment, we notice that the translated log-weights are representable by the floating point arithmetic of the system.

I.4 Bootstrap particle filter for the stochastic volatility model

The stochastic volatility model is represented as

$$
\begin{aligned}
x_t \mid x_{t-1} &\sim N(x_t; \phi x_t, \sigma^2) \\
y_t \mid x_t &\sim N(y_t; 0, \beta^2\ exp(x_t)) \\
\theta &= \{\phi, \sigma, \beta\}
\end{aligned}
$$

Assuming $\theta = \{0.98, 0.16, 0.7\}$, estimate the marginal filtering distribution at each time index $t = 1, ..., T$ using the bootstrap particle filter with $N = 500$ particles.

Initial experiment to estimate the initial state

```
x0 <- 0
T <- 1000000
x <- rep(0, T)
x[1] <- 0.98 * x0 + 0.16*rnorm(1)

for (t in 2:T) {
  x[t] <- 0.98 * x[t-1] + 0.16*rnorm(1)
}
mean(x[5000:T])
```

```
## [1] -0.001071948
```

```
sd(x[5000:T])
```

```
## [1] 0.8075554
```

The estimated mean is close to zero, and SD is 0.8 So, we assume that $x_0 \sim N(0, .8)$

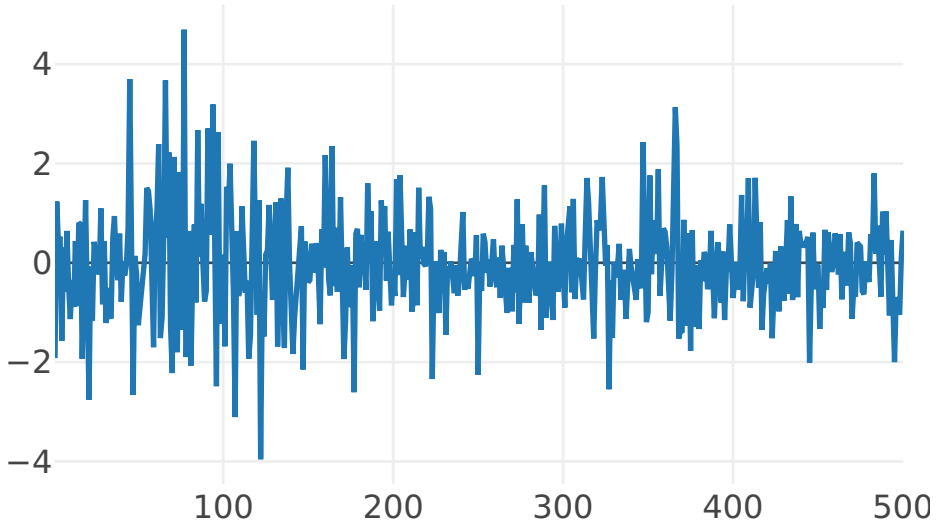
A simple implementation of bootstrap particle filter for volatility model.

```r
# Bootstrap particle filter for volatility model
volatilityModelBPF <- function(param, x0, y, N = 100) {
  T <- length(y) # number of states
  #Initialize the parameters
  A <- param[1] # coefficient
  Q <- param[2] # process noise
  beta <- param[3] # contributes in measurement noise

  # define variables
  particles <- matrix(0, nrow = N, ncol = T)
  normalisedWeights <- matrix(0, nrow = N, ncol = T)
  xHatFiltered <-rep(0, T)

  # Initialize variables for state 1
  particles[ ,1] <- A*x0 + sqrt(Q)*rnorm(N)

  # Weighting step
  weights <- dnorm(y[1], mean = 0, sd = sqrt(beta) * exp(particles[, 1]/2))
  # Normalize weights
  sum_weight <- sum(weights)
  normalisedWeights[, 1] <- weights / sum(weights)
  # Estimate the state
  xHatFiltered[1] <- sum(particles[, 1]*normalisedWeights[, 1])

  for (t in 2:T) {
    # Resampling step
    newAncestors <- multinomial.resample(normalisedWeights[, t - 1])

    # Propagation step
    particles[, t] <- A * particles[newAncestors, t-1] +
      sqrt(Q) * rnorm(N)

    # Weighting step
    weights <- dnorm(y[t], mean = 0, sd = sqrt(beta) * exp(particles[, t]/2))

    # Normalize weights
    sum_weight <- sum(weights)
    normalisedWeights[, t] <- weights / sum(weights)

    # Estimate the state
    xHatFiltered[t] <- sum(particles[, t]*normalisedWeights[, t])
  }

  return(list(xHatFiltered = xHatFiltered,
              particles = particles,
              normalisedWeights = normalisedWeights))

}
```

Applying the BPF on the dataset

```r
#Read the observations and plot them
y <- read.csv("seOMXlogreturns2012to2014.csv", header=FALSE )
```

```
y <- y[,1]
T <- length(y)
N <- 500
plot_ly(x = c(1:T), y = y,
        name = 'Simulated States', type = 'scatter', mode = 'lines')
```




```
# Call BPS to estimate the states
param <- c(0.98, 0.16^2, 0.70^2)
x0 <- 0.8 * rnorm(N)
result = volatilityModelBPF(param, x0, y, N = N)

plot_ly(x = c(1:T), y = result$xHatFiltered,
            name = 'Simulated States', type = 'scatter', mode = 'lines')
```