DD2447 Statistical Methods in Applied Computer Science
Assignment 2

Diar Sabri 19940227-4196, Navid Haghshenas 19940610-5057

15 January 2020

1 Gibbs sampler for the magic word

The objective is to estimate our desired starting positions r_n of "motifs" or "magic words" using Gibbs sampler from N sequences of length M & motif length W, given the parameters α' =alphaBg & α =alphaMw. K is the length of the alphabet, which in our case is 4 (for both synthetic data & given DNA-data).

Each sequence contains one motif (or magic word) and the remaining content of the sequence is labled as the background. Furthermore, each starting position is randomly picked in the range $[M - W + 1]$ uniformely. Then the $j : th$ positions in the magic word are sampled from $q_j(x) = Cat(x|\Theta_j)$ where Θ_j has a $Dir(\Theta|\alpha^{'})$ prior. All other positions in the sequences are sampled from the background distribution $q(x) = Cat(x|\Theta)$ where Θ has a $Dir(\Theta \mid \alpha)$ prior.

The goal is to obtain the posterior $p(r_1, ..., r_N|D)$ where D is the set of sequences generated from our created model & r_n, the motif start positions. Using the marginal likelihoods, the Gibbs sampler is implemented which gives us the desired posterior with high accuracy. The code below illustrates the pseudo code for the main function:

```
function gibbsSampler:
    For iter in Iterations:
        For n in N:
            prob = P(r[n]|R[(-)n]) #estimate the probability for motif & background positions
            Normalize(prob)
            Sample over r[n] #sample from distributed motif & bg positions
            Refresh R
        END loop
    END loop
    Return list of final guesses (one chain)
```

Of course, there are other functions and calculations that are calculating the variable *prob*. The entire code can be found in appendix.

Question 1

The first sub-question is to estimate the posterior using our Gibbs sampler with evidence. The plots below are for N=25, W=10, M=100, chains=10 & iterations=20. For our inputs, we have $\alpha' = [1, 1, 1, 1]$ and $\alpha = [0.8, 0.8, 0.8, 0.8]$ as requested by the course exterminator. The plot to the right represents the histogram for the motif on index 24 i.e. the last motif. It is clear that number ... gets the highest vote and is chosen as our guessed initial position. All plots for the synthetic data can be found in the appendix. The plot on the left displays the convergence towards the original initial positions of the motifs where each color represents one sequence / motif. We can see that the guesses are struggling to find a value at first and are deviating from the actual result (y-axis. After a few iterations (x-axis) we have almost all sequences converging towards their demanded values.

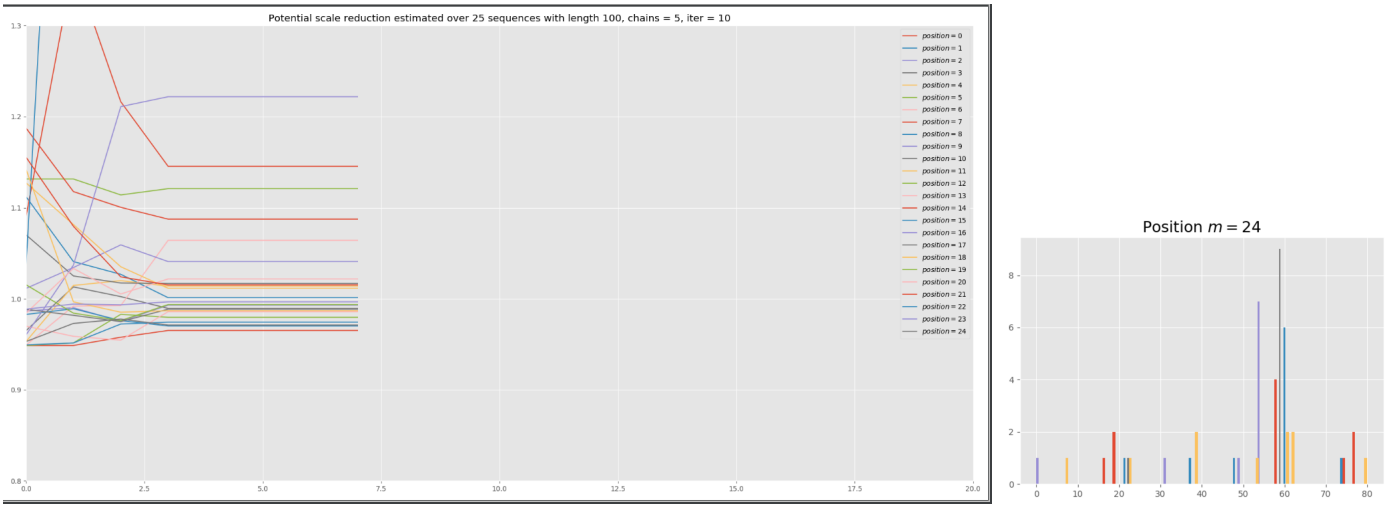


Figure 1: Histogram of motif=25 (index 24) & the over time- convergence for all 25 sequences.

$Original = [58, 73, 46, 59, 53, 54, 38, 90, 48, 80, 82, 56, 25, 12, 35, 20, 48, 16, 11, 62, 33, 27, 75, 40, 54]$
$Guessed = [59, 72, 46, 58, 53, 54, 37, 90, 48, 80, 82, 55, 25, 11, 4, 20, 49, 16, 11, 28, 33, 28, 76, 40, 54]$

Question 2

For this part, we had to read the given dna text file and generate random, original positions in the same way as the synthetic data. The differences here are that we do not need to generate data from scratch and that the data is much bigger (N=25, W=10, M=1000, chains=10 & iterations=20. For our inputs, we have the same α' and α as question 1.

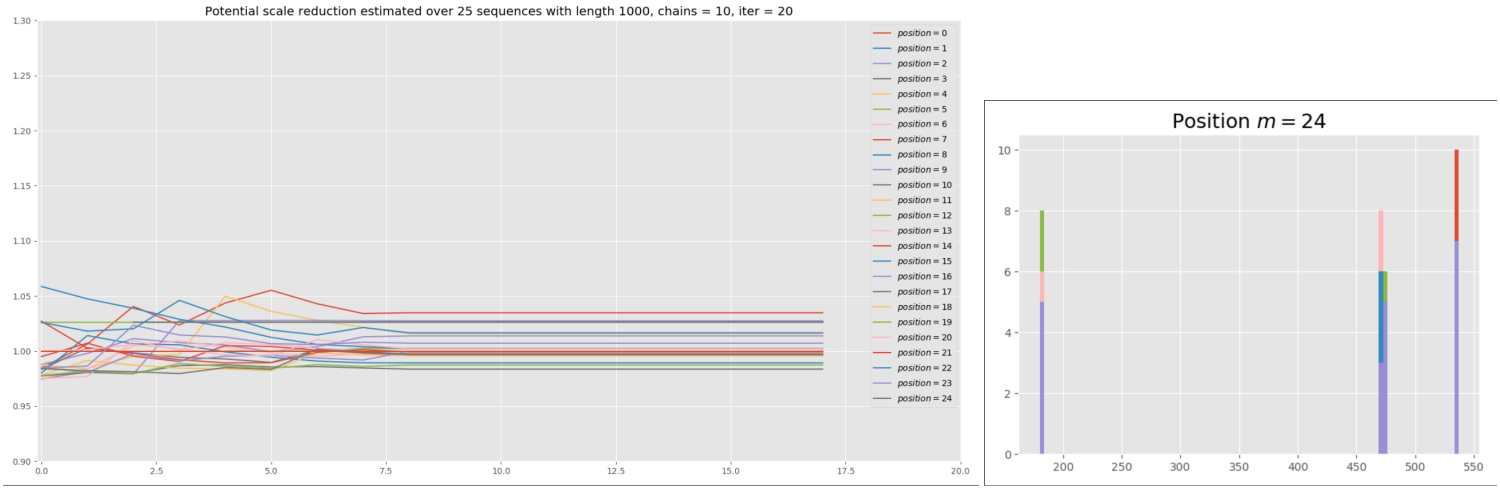


Figure 2: Histogram of motif=25 (index 24) for DNA & the over time- convergence for all 25 sequences.

The accuracy of the given DNA data is not as good as the accuracy for synthetic data. The calculations disregard many of the positions but they are still converged with the Gibbs sampler to a trivial initial position in range $[M - W + 1]$, as requested. The rest of the figures for this implementation can be found in the provided appendix together with the code, which is identical to the one for question 1 but with the dna.txt as the input.

2 SMC for the stochastic volatility model

Question 3

To generate the parameters and data we followed the instructions according to the problem description. Given information: $T = 100, \phi = 1.0, \sigma = 0.16, \beta = 0.64$. The stochastic volatility model is defined as

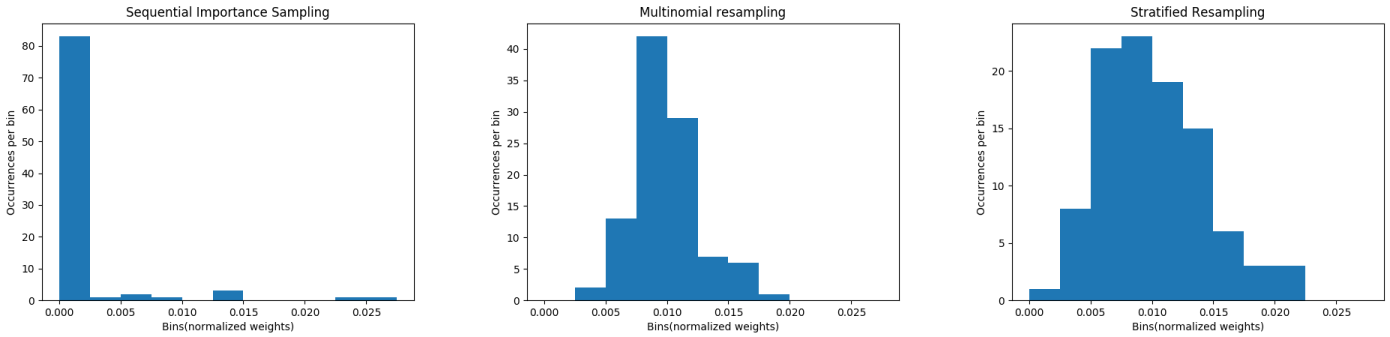
$$
\begin{aligned}
X_1 &\sim \mathcal{N}(x_1|0, \sigma^2) \\
X_t|(X_{t-1} = x_{t-1}) &\sim \mathcal{N}(x_t|\phi x_{t-1}, \sigma^2), && t = 1, \ldots, T, \\
Y_t|(X_t = x_t) &\sim \mathcal{N}(y_t|0, \beta^2 \exp(x_t)), && t = 1, \ldots, T,
\end{aligned}
$$

Figure 2: Stochastic Volatility Model

Something to note regarding the generation of data is that normal distribution in numpy takes the standard deviation as an argument, while it is declared via the variance in the assignment.

Question 4

The justification for our choice of proposal stems from the fact of trying to simplify the weight update function. We want to use the observed Y_T to infer X_T. We can extract the probability and weights for each sequence by drawing X_t given Y_t, and thereafter estimating X_T with the help of a sequence. In practice, we have a base case for $t = 0$ and then using the mentioned simplification for $t > 0$. Most of the underlying algorithms in this section have taken an inspiration from the lectures and the pseudo-code available. The point estimate $\hat{x_T}$ is calculated by summing the normalized weights in the last iteration multiplied with X in the last iteration.



Algorithm 1 Sequential Importance Sampling

1: **Initialize for t=1**
2: draw $x_1^i \sim p(x_1)$
3: compute weights $\tilde{w}_1^i = p(y_1 \mid x_1^i)$ and normalize $w_1^i = \frac{\tilde{w}_1^i}{\sum_j \tilde{w}_1^j}$
4: **for** $t = 2$ to number of states T **do**
5: Draw $x_t^i \sim p(x_t \mid x_{t-1}^i)$
6: compute weights $\tilde{w}_t^i = p(y_t \mid x_t^i)\tilde{w}_{t-1}^i$ and normalize $w_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j}$
7: **end for**

Figure 3: Pseudo.code of the Sequential Importance Sampling algorithm

The empirical variance and the point estimate values are both using 100 particles (for all three methods, SIS/ Multinomial/ Stratified).

$\hat{x_T} = -0.3734$

Empirical variance = 0.001199

# particles	Mean Squared Error
100	0.2246
200	0.5785
400	0.5735
800	0.5111

Table 1: Mean squared errors for Sequential Importance Sampling

Question 5

In this section, we build upon the SIS-algorithm and implement a bootstrap particle filter with a multinomial resampling method. The algorithm again is based on the lectures, and the resampling method is simplified by the use of the numpy package.

Algorithm 2 Bootstrap Particle Filter

1: **Initialize for t=1**
2: draw $x_1^i \sim p(x_1)$
3: compute weights $\tilde{w}_1^i = p(y_1 \mid x_1^i)$ and normalize $w_1^i = \frac{\tilde{w}_1^i}{\sum_j \tilde{w}_1^j}$
4: **for** $t = 1$ to number of states T **do**
5: resample $\{x_{t-1}^i, w_{t-1}^i\}$ to obtain N equally weighted particles $\{\bar{x}_{t-1}^i, \frac{1}{N}\}$
6: Draw $x_t^i \sim p(x_t \mid \bar{x}_{t-1}^i)$
7: compute weights $\tilde{w}_t^i = p(y_t \mid x_t^i)$ and normalize $w_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j}$
8: **end for**

Figure 4: Pseudo.code of the bootstrap particle filter

$\hat{x_T} = -0.3184$

Empirical variance $= 7.8703e - 06$

# particles	Mean Squared Error
100	0.4201
200	0.0434
400	0.1708
800	0.4895

Table 2: BPF with multinomial resampling

Question 6

For the BPF with stratified resampling, an inspiration was taken from the lecture on SIS by Arnaud Doucet from the University of British Columbia [1].

$\hat{x_T} = -0.2618$

Empirical variance $= 1.8892e - 05$

# particles	Mean Squared Error
100	0.2099
200	0.1045
400	0.1140
800	0.4288

Table 3: BPF with stratified resampling

Question 7

The histogram of SIS show that almost all weights are in the first bin, and the rest of the bins contain almost no weights. The resampling methods show a completely different histogram, indicating significantly more spread across the different bins. The resampling methods can according to the course lectures reduce the empirical variance, which is in line with our results. Additionally, all the bins are very close to zero (with the highest being 0.025), and SIS is almost completely in the first (0) bin.

References

[1] Arnaud Doucet. Sequential importance sampling resampling. *https://www.cs.ubc.ca/ arnaud/samsi/samsi¡ec3.pdf*, 2019.

3 Appendix

3.1 Gibbs sampler for the magic word

```
from __future__ import division

import numpy as np
from collections import Counter
import math as math
import matplotlib.pyplot as plt
import tqdm as tqdm

plt.style.use('ggplot')


def dna(numSeq, lenSeq, lenMotif):
    seqList = open("data.txt", "r").read()
    seqList = [item.split(" ") for item in seqList.split('\n')[:numSeq]]
    for j in seqList:
        del j[(lenSeq - 1):999]

    startList = []
    for s in range(numSeq):
        r = np.random.randint(0, lenSeq - lenMotif + 1)
        startList.append(r)
    print('Start list: ', startList)
    return seqList, startList

def generate_sequences(alphabet, alphaMw, alphaBg, N, M, W):
    thetaBg = np.random.dirichlet(alphaBg)
    thetaMw = np.random.dirichlet(alphaMw, W)

    samples = []
    positions = []
    for i in range(N):
        seq = []
        pos = np.random.randint(0, M - W + 1)  # Mw randomize init positions
        for j in range(M):
            if pos <= j and j < pos + W:  # Mw sampling
                seq.append(sampleIt(alphabet, thetaMw[j - pos]))
            else:  # bg sampling
                seq.append(sampleIt(alphabet, thetaBg))

        samples.append(seq)
        positions.append(pos)
    print('Start list: ', positions)

    return samples, positions

def sampleIt(alphabet, cat):
    return alphabet[np.argmax(np.random.multinomial(1, cat))]

def estimate_prior(alphabet, alphaMw, alphaBg, N, M, W, data, positions):
    B = N * (M - W)
```

```
    # estimating background:
    bgData = [data[n][0:positions[n]] + data[n][positions[n] + W:] for n in range(N)]
    Bg = Counter([token for seq in bgData for token in seq])

    alphaBgSum = sum(alphaBg)
    calc = math.lgamma(alphaBgSum) - math.lgamma(B + alphaBgSum)

    probs = [math.lgamma(Bg[alphabet[k]] + alphaBg[k]) - math.lgamma(alphaBg[k]) for k in range(K)]

    p = calc + sum(probs)

    # estimating magic word:
    magicWordData = [data[n][positions[n]:positions[n] + W] for n in range(N)]
    magicWordData = np.array(magicWordData)

    N_j = [Counter(magicWordData[:, w]) for w in range(W)]

    alphaSum = sum(alphaMw)
    calc = math.lgamma(alphaSum) - math.lgamma(N * W + alphaSum)

    probsj = []
    for j in range(W):
        probskOverj = [math.lgamma(N_j[j][alphabet[k]] + alphaMw[k]) - math.lgamma(alphaMw[k]) for k
                       in range(K)]

        probskOverj = calc + sum(probskOverj)
        probsj.append(probskOverj)

    return p, probsj


def estPosterior(alphabet, alphaMw, alphaBg, sequence_id, M, W, data, prev_positions):
    pos_proba = []
    for r_i in range(M - W):
        positions = list(prev_positions)
        positions[sequence_id] = r_i

        p_b, p_mw = estimate_prior(alphabet, alphaMw, alphaBg, N, M, W, data, positions)
        p = p_b + sum(p_mw)
        pos_proba.append(p)

    return pos_proba

def p_normalize(p):
    p = np.exp(p - np.max(p))
    p = p/np.sum(p)
    return p

def gibbsSampler(alphabet, data, alphaMw, alphaBg, W, itrr):

    iterations = itrr
    samples = []
    initPos = np.random.randint(0, M - W + 1, size=N)

    samples.append(initPos)

    for _ in tqdm.tqdm(range(iterations)):
        positions = []
        gibbsCondition = samples[-1]
```

```
        for n in range(N):
            posProb = estPosterior(alphabet, alphaMw, alphaBg, n, M, W, data, gibbsCondition)
            posProb = p_normalize(np.asarray(posProb))
            multSamp = np.random.multinomial(1, posProb)

            position = np.argmax(multSamp)

            gibbsCondition[n] = position
            positions.append(position)

        samples.append(np.array(positions))
    # lag
    chain = [samples[j] for j in range(0, iterations)]

    return chain


def estPotentialScaleReduction(data):
    '''
    Returns a float representing the estimated PSR for given data.
    '''
    Chains, Iter, Dimension = data.shape

    yAx1 = np.mean(data, axis=1)

    yAx2 = np.mean(yAx1, axis=0)

    B = Iter / (Chains - 1.) * np.sum((yAx1 - yAx2) ** 2, axis=0)

    W = np.sum(np.array([(data[c, :, :] - yAx1[c, :]) ** 2 for c in range(Chains)]), axis=1)
    W = 1. / Chains * np.sum(1. / (Iter - 1) * W, axis=0)

    V = (Iter - 1.) / Iter * W + 1. / Iter * B

    psr = np.sqrt(V / W)

    return psr


if __name__ == '__main__':
    cat = np.ones(4) * 1 / 4
    alphabet = ['a', 'b', 'c', 'd']
    M = 100
    W = 10
    N = 25
    K = len(alphabet)

    alphaBg = [1, 1, 1, 1]
    alphaMw = [0.8, 0.8, 0.8, 0.8]

    data, positions = generate_sequences(alphabet, alphaMw, alphaBg, N, M, W) #
    #data, positions = dna(N, M, W)

    chains = 5
    iter = 10
    gibbs = np.array([gibbsSampler(alphabet, data, alphaMw, alphaBg, W, itrr=iter) for _ in range(chains)])

    #find most guessed guesses!
    axis=0
```

```
u, index = np.unique(gibbs, return_inverse=True)
print(u[np.argmax(np.apply_along_axis(np.bincount, axis, index.reshape(gibbs.shape), None, np.max(index)
+ 1), axis=axis)])

#histograms
for motif in range(N):  # for each position
    plt.plot()
    for c in range(chains):
        plt.hist(gibbs[c, :, motif], label='$c={}$'.format(c), bins=100)
        plt.title('Position $m={}$'.format(motif), fontsize=18)
    plt.show()

#"convergogram"
Chains, Iter, Data = gibbs.shape
convergence = np.array([estPotentialScaleReduction(gibbs[:][0:t][:]) for t in range(2, Iter)])

plt.figure(figsize=(20, 10))
plt.plot()
for pos in range(Data):  # for each position plot a new graph
    plt.plot(convergence[:,pos], label='$position = {}$'.format(pos))
plt.title('Potential scale reduction estimated over {} sequences with length {}, chains = {}
, iter = {}'.format(N,M,chains,iter))
plt.legend(loc=1)
plt.axis([0,20,0.8,1.3])
plt.show()
```

3.2 SMC for the stochastic volatility model

```
import numpy as np
import pdb
import matplotlib.pyplot as plot
import scipy.stats
import math

def main():
    phi = 1.0
    sigma = 0.16
    beta = 0.64
    T = 100

    seed = 57832
    np.random.seed(seed)

    particles = [100,200,400,800]
    for p in particles:
        xT,y = generateData(T,phi,sigma,beta)
        data = list()
        print('\nNumber of particles: ',p)
        #SIS
        w1, x1 = sequentialImportanceSampling(T, y, p, phi, sigma, beta)
        data.append(w1[-1]) #weight vector of last iteration
        xx_1 = np.divide(w1[T-1],np.sum(w1[T-1]))
        print('\nx_hat1',round(sum(x1[T-1]*xx_1),8))
        print('var1',np.var(xx_1))
        print('MSE1',round(np.sum(np.multiply(xx_1, np.power(x1[-1] - xT[-1], 2))),8))

        #Multinomial resampling
        w2, x2 = BPF_multinomialResampling(T, y, p, phi, sigma, beta)
        data.append(w2[-1]) #weight vector of last iteration
        xx_2 = np.divide(w2[T-1],np.sum(w2[T-1]))
        print('\nx_hat2',round(sum(x2[T-1]*xx_2),8))
        print('var2',np.var(xx_2))
        print('MSE2',round(np.sum(np.multiply(xx_2, np.power(x2[-1] - xT[-1], 2))),8))

        #Stratified resampling
        w3, x3 = BPF_stratifiedResampling(T, y, p, phi, sigma, beta)
        data.append(w3[-1]) #weight vector of last iteration
        xx_3 = np.divide(w3[T-1],np.sum(w3[T-1]))
        print('\nx_hat3',round(sum(x3[T-1]*xx_3),8))
        print('var3',np.var(xx_3))
        print('MSE3',round(np.sum(np.multiply(xx_3, np.power(x3[-1] - xT[-1], 2))),8))

    #Plots
    bs = np.arange(0,0.03,0.0025)
    plot.hist(data[0],bins=bs)
    plot.title('Sequential Importance Sampling')
    plot.xlabel('Bins(normalized weights)')
    plot.ylabel('Occurrences per bin')
    plot.show()

    plot.hist(data[1],bins=bs)
    plot.title('Multinomial resampling')
    plot.xlabel('Bins(normalized weights)')
    plot.ylabel('Occurrences per bin')
    plot.show()
```

```
    plot.hist(data[2],bins=bs)
    plot.title('Stratified Resampling')
    plot.xlabel('Bins(normalized weights)')
    plot.ylabel('Occurrences per bin')
    plot.show()

def generateData(T,phi,sigma,beta):
    x = np.zeros(T)
    y = np.zeros(T)
    x[0] = np.random.normal(0,sigma)
    y[0] = np.random.normal(0,math.sqrt((beta**2)*(math.e**x[0])))
    for t in range(1, T):
        x[t] = np.random.normal(phi * x[t-1], sigma)
        y[t] = np.random.normal(0, math.sqrt((beta**2)*(math.e**x[t])))
    return x, y

def sequentialImportanceSampling(T, y, particles, phi, sigma, beta):
    x = np.zeros([T, particles])
    w = np.zeros([T, particles])

    for n in range(particles):
        x[0][n] = np.random.normal(0,sigma)
        w[0][n] = scipy.stats.norm.pdf(y[0], scale = math.sqrt((beta**2)*(math.e**x[0,n])))
    w[0] = w[0]/np.sum(w[0])

    for t in range(1,T):
        for n in range(particles):
            x[t][n] = np.random.normal(phi * x[t-1][n], sigma)
            w[t][n] = scipy.stats.norm.pdf(y[t], scale = math.sqrt((beta**2)*(math.e**x[t,n])))* w[t-1][n]
        w[t] = w[t]/np.sum(w[t]) #normalize weights
    return w, x

def BPF_multinomialResampling(T, y, particles, phi, sigma, beta):
    x = np.zeros([T, particles])
    w = np.zeros([T, particles])

    for n in range(particles):
        x[0][n] = np.random.normal(0, sigma)
        w[0][n] = scipy.stats.norm.pdf(y[0], scale = math.sqrt((beta**2)*(math.e**x[0,n])))
    w[0] = w[0]/np.sum(w[0])

    for t in range(1,T):
        for n in range(particles):
            resampling = np.argmax(np.random.multinomial(1, w[t-1]))
            x[t][n] = np.random.normal(phi * x[t-1][resampling], sigma)
            w[t][n] = scipy.stats.norm.pdf(y[t], scale = math.sqrt((beta**2)*(math.e**x[t,n])))
        w[t] = w[t] /np.sum(w[t]) #normalize weights
    return w, x

def BPF_stratifiedResampling(T, y, particles, phi, sigma, beta):
    x = np.zeros([T, particles])
    w = np.zeros([T, particles])

    u1 = np.random.uniform(low=0, high = 1/particles)
    u = np.array([u1 + (i-1)/particles for i in range(particles)])

    for t in range(T):
        if t > 0:
```

```
            count = 0
        for n in range(particles):
            if t == 0:
                x[t][n] = np.random.normal(0, sigma)
                w[t][n] = scipy.stats.norm.pdf(y[t], scale = math.sqrt((beta**2)*(math.e**x[t,n])))
                w[0] = w[0]/np.sum(w[0])
            else:
                lower = np.sum(w[t-1][:n])
                higher = np.sum(w[t-1][:n+1])
                strat = np.sum((lower <= u) & (u < higher))
                if strat != 0:
                    for k in range(strat):
                        x[t][k+count] = np.random.normal(phi * x[t-1][n], sigma)
                        w[t][k+count] = scipy.stats.norm.pdf(y[t], scale = math.sqrt((beta**2)
                        *(math.e**x[t,k+count])))* w[t-1][n]
                    count += strat
        #normalize
        w[t] = w[t]/np.sum(w[t])
    return w,x

if __name__ == '__main__':
    main()
```