Probabilistic Graphical Models - Tutorial 7

Fernando García Sanz

February 17, 2020

2 Classical Monte Carlo sampling methods

Task 2.1: Rejection Sampling

Task (a):

Because knowing that $p(x) < Mq(x)$, being M a constant $M < \infty$ is enough. It is not necessary to know either the shape of the probability $p(x)$ nor if it actually is a probability function. The condition for accepting one point using *rejection sampling* is that the constant M times $q(x)$ has to be greater than the value $p(x)$. The following image is a representation of rejection sampling:

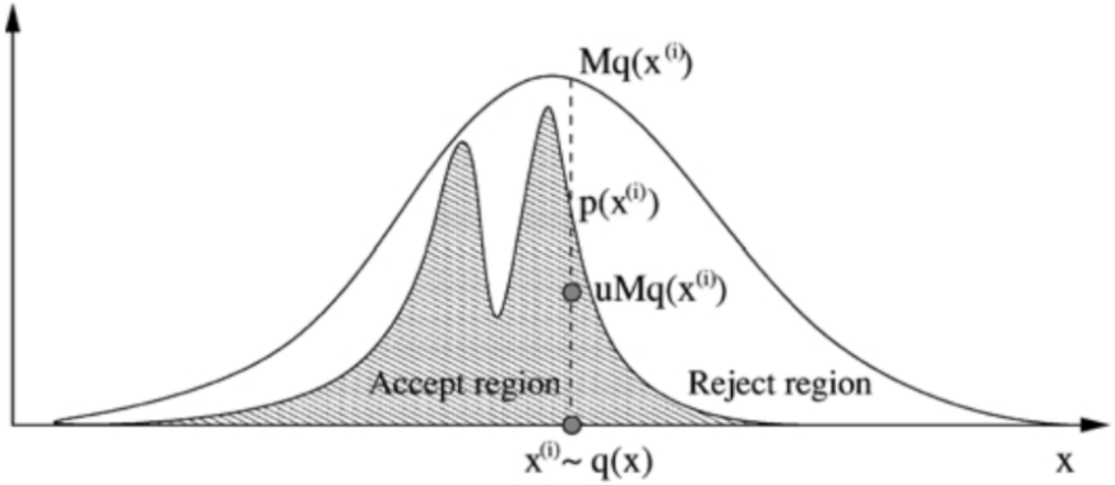

Figure 1: Rejection sampling representation.

Task (b):

$$p\left(u < \frac{\tilde{p}(x)}{Mq(x)}\right) = \sum_x p\left(u < \frac{\tilde{p}(x)}{Mq(x)}, x\right) = \sum_x p\left(u < \frac{\tilde{p}(x)}{Mq(x)} \mid x\right)\ p(X = x)$$

Considering that $u \sim U(0,1)$ and that the only accepted x values are those that fulfill the condition determined in the above expression, it is possible to say that $p(accept \mid x) = \frac{\tilde{p}(x)}{Mq(x)}$. Moreover, the fact of $x \sim q(x)$ implies that $p(X = x)$ is equal to $q(x)$. Therefore, continuing the previous expression:

$$p\left(u < \frac{\tilde{p}(x)}{Mq(x)}\right) = \sum_x \frac{\tilde{p}(x)}{Mq(x)}\ q(x) = \frac{1}{M} \sum_x \tilde{p}(x)$$

From here, knowing that $p(x) = \frac{\tilde{p}(x)}{Z}$, it can be deduced that $Z * p(x) = \tilde{p}(x)$, being $p(x)$ a probability density function. So, finally:

$$p\left(u < \frac{\tilde{p}(x)}{Mq(x)}\right) = \frac{1}{M}\sum_x \tilde{p}(x) = \frac{1}{M}\sum_x p(x)Z = \frac{Z}{M}$$

This is obtained since summing over all possible values of x in a probability density function returns the entire probability, which is always 1.

Task 2.2: Importance Sampling

Task (a):

Moment matching allows to write:

$$\mathbb{E}_{p(x)}[f(x)] = \int_x f(x)p(x)dx$$

By means of the *Strong Law of Large Numbers*, which tells us that along an infinite succession of random variables, their average value converges to the value of μ, it can be stated that:

$$\mathbb{E}_{p(x)}[f(x)] \approx \frac{1}{L}\sum_{l=1}^{L} f(x^{(l)})$$

Furthermore, applying the fundamental identity of *Importance Sampling*, under the condition of $supp(q) \supset supp(p)$, this expression converges to:

$$\begin{aligned}\frac{1}{L}\sum_{l=1}^{L} f(x^{(l)}) = \frac{1}{L}\sum_{l=1}^{L} f(x^{(l)})\frac{q(x^{(l)})}{q(x^{(l)})} &\approx \sum_{l=1}^{L} f(x^{(l)})\frac{q(x^{(l)})}{q(x^{(l)})}p(x^{(l)}) \\ = \sum_{l=1}^{L} f(x^{(l)})q(x^{(l)})w(x^{(l)}) &\approx \frac{1}{L}\sum_{l=1}^{L} f(x^{(l)})w(x^{(l)})\end{aligned}$$

Task (b):

Importance Sampling makes use of an approximation to get the average value by means of the *Strong Law of Large Numbers* (explained in the previous task). The weights are a method of reducing the effect of the biased distribution when $w(x) := \frac{p(x)}{q(x)}$ is calculated. A high weight increases the amount of samples within regions of special interest, and vice versa for a low one.

3 Metropolis-Hastings Sampling

Task 3.1: Pen-and-paper

Task (a):

The expression that needs to be proven is the following:

$$p(x^{(i+1)})T(x^{(i)}|x^{(i+1)}) = p(x^{(i)})T(x^{(i+1)}|x^{(i)})$$

Firstly, the following expressions are known:

$$\begin{aligned}T(x^{(i+1)}|x^{(i)}) &= q(x^{(i+1)}|x^{(i)})A(x^{(i)}, x^{(i+1)}) + \delta_{x^{(i)}}(x^{(i)})r(x^{(i)}) \\ A(x^{(i)}, x^{(i+1)}) &= min\left\{1, \frac{p(x^{(i+1)})q(x^{(i)}|x^{(i+1)})}{p(x^{(i)})q(x^{(i+1)}|x^{(i)})}\right\} \\ r(x^{(i)}) &= \int_x q(x^{(i+1)}|x^{(i)})[1 - A(x^{(i)}, x^{(i+1)})]dx^{(i+1)}\end{aligned}$$

Analyzing the expression A, there are two possible outcomes, either the minimum is one, or the minimum is the fraction, so the first term of the equation $T(x^{(i+1)}|x^{(i)})$ will be:

$$\text{Case: 1 is the minimum} \implies q(x^{(i+1)}|x^{(i)})$$

$$\text{Case: 1 is the maximum} \implies q(x^{(i+1)}|x^{(i)})\,\frac{p(x^{(i+1)})q(x^{(i)}|x^{(i+1)})}{p(x^{(i)})q(x^{(i+1)}|x^{(i)})} = q(x^{(i)}|x^{(i+1)})\,\frac{p(x^{(i+1)})}{p(x^{(i)})}$$

From here, it is known that:

$$q(x^{(i+1)}|x^{(i)})A(x^{(i)},x^{(i+1)}) = min\left\{q(x^{(i+1)}|x^{(i)}), q(x^{(i)}|x^{(i+1)})\,\frac{p(x^{(i+1)})}{p(x^{(i)})}\right\}$$

Which can be transformed, for convenience, into:

$$q(x^{(i+1)}|x^{(i)})A(x^{(i)},x^{(i+1)}) = \frac{1}{p(x^{(i)})}\,min\left\{q(x^{(i+1)}|x^{(i)})\,p(x^{(i)}), q(x^{(i)}|x^{(i+1)})\,p(x^{(i+1)})\right\}$$

The *min* function is symmetric with respect to the change of $x^{(i)}$ by $x^{(i+1)}$ when multiplying by the factor $p(x^{(i)})$, which allows the following transformation:

$$= p(x^{(i)})\frac{1}{p(x^{(i)})}\,min\left\{q(x^{(i+1)}|x^{(i)})\,p(x^{(i)}), q(x^{(i)}|x^{(i+1)})\,p(x^{(i+1)})\right\}$$
$$= p(x^{(i+1)})\frac{1}{p(x^{(i+1)})}\,min\left\{q(x^{(i)}|x^{(i+1)})\,p(x^{(i+1)}), q(x^{(i+1)}|x^{(i)})\,p(x^{(i)})\right\}$$

Now, analyzing the right part of the sum of $T(x^{(i+1)}|x^{(i)})$, the following can be derived:

$$\delta_{x^{(i)}}(x^{(i)})r(x^{(i)}) = \delta_{x^{(i)}}(x^{(i)})\int_x q(x^{(i+1)}|x^{(i)})[1-A(x^{(i)},x^{(i+1)})]dx^{(i+1)}$$
$$= \delta_{x^{(i)}}(x^{(i)})\left[\int_x q(x^{(i+1)}|x^{(i)})dx^{(i+1)} - \int_x q(x^{(i+1)}|x^{(i)})A(x^{(i)},x^{(i+1)})dx^{(i+1)}\right]$$

The left integral integrates over the values of x, so the result of it will be equal to 1, obtaining, therefore, the following expression:

$$\delta_{x^{(i)}}(x^{(i)})r(x^{(i)}) = \delta_{x^{(i)}}(x^{(i)})\left[1-\int_x q(x^{(i+1)}|x^{(i)})A(x^{(i)},x^{(i+1)})dx^{(i+1)}\right]$$

As stated before, it can be proven that the terms enclosed in the integral are symmetric with respect to the exchange between the elements $x^{(i)}$ and $x^{(i+1)}$.

Task (b):

The fact of working with an asymmetric proposal distribution may be motivated by the fact that the original distribution has an asymmetric shape. It is possible to identify the symmetry of a distribution by analyzing its *skewness* property. It can happen, for example, during the estimation of the posterior distribution of variance, where the values are constrained to only positive values, and not symmetric ones. If the values are bounded by some kind of constraint, it is convenient to use an asymmetric distribution.

Task (c):

In *Independence Sampler*, a special case of *Metropolis Hastings* algorithm, the proposal distribution is independent of the current state, i.e. $q(x^{(*)}|q(x^{(i)})) = q(x^{(*)})$. Therefore, if we want to sample from $p(x|y)$, knowing that $p(y,x) = p(x)p(y|x)$:

$$A = min\left\{1, \frac{p(x^{(*)})q(x^{(i)})}{p(x^{(i)})q(x^{(*)})}\right\} \implies \frac{p(x^{(*)})p(y,x^{(i)})}{p(x^{(i)})p(y,x^{(*)})} = \frac{p(x^{(*)})p(x^{(i)})p(y|x^{(i)})}{p(x^{(i)})p(x^{(*)})p(y|x^{(*)})} = \frac{p(y|x^{(i)})}{p(y|x^{(*)})}$$

This is a good choice since $p(y|x)$ encloses $p(x|y)$ (not normalized case) and the probability of the observations is easier to find.

Task 3.2: Coding Exercise

Code can be found in the appendix.

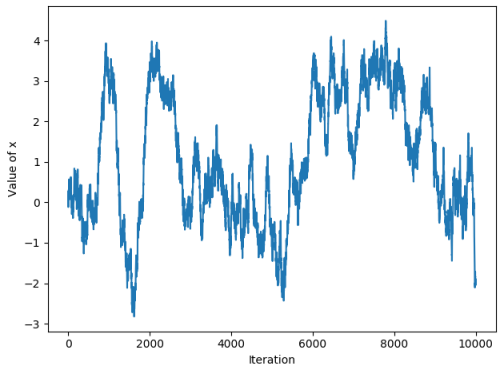

Figure 2: Metropolis-Hastings random walk with $\sigma_{proposal} = 0.1$. 10000 iterations.

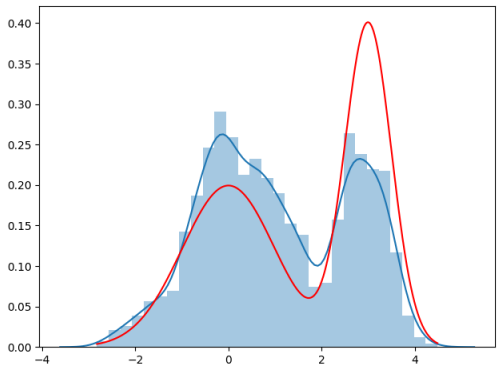

Figure 3: Metropolis-Hastings distribution with $\sigma_{proposal} = 0.1$. 10000 iterations.

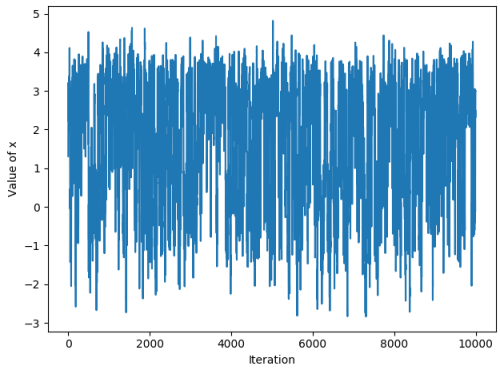

Figure 4: Metropolis-Hastings random walk with $\sigma_{proposal} = 1$. 10000 iterations.

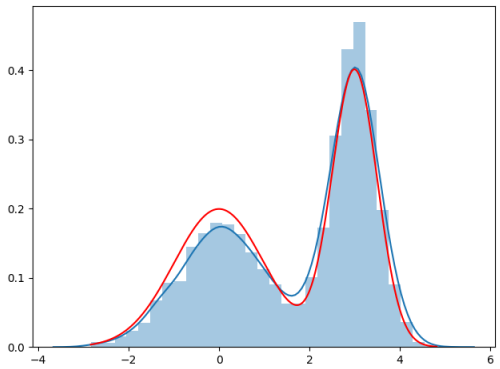

Figure 5: Metropolis-Hastings distribution with $\sigma_{proposal} = 1$. 10000 iterations.

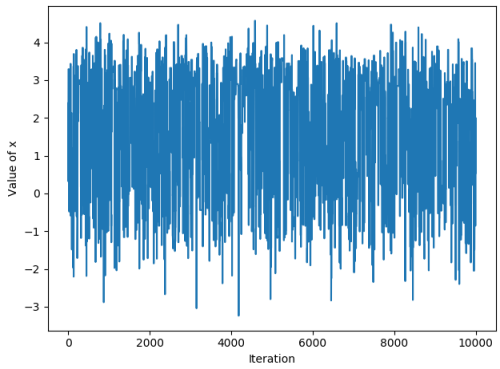

Figure 6: Metropolis-Hastings random walk with $\sigma_{proposal} = 10$. 10000 iterations.

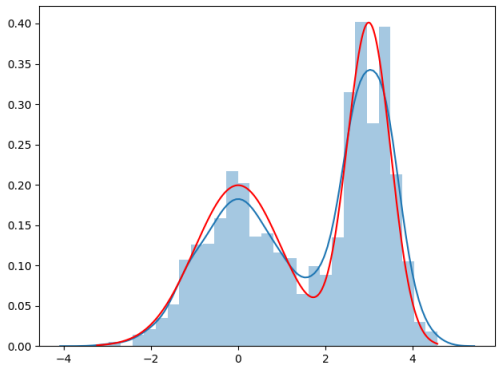

Figure 7: Metropolis-Hastings distribution with $\sigma_{proposal} = 10$. 10000 iterations.

As it can be seen, the greater the variance of the proposal distribution, the bigger the difference between the x values of the random walk during the iterations. The best approximation between the predicted (blue) and the real (red), can be found in figure 5.

4 Particle Filtering / Sequential Monte Carlo (SMC)

Task 4.1: Analytic Exercise

The following expressions are known about the weights:

$$\tilde{w}_t^{(i)} = \frac{w(x_{0:t}^{(i)})}{\sum_{j=1}^{N} w(x_{0:t}^{(j)})} \text{ : norm. importance weights} \qquad w(x_{0:t}^{(i)}) = \frac{p(x_{0:t}^{(i)}|y_{1:t})}{\pi(x_{0:t}^{(i)}|y_{1:t})} \text{ : imp. weights}$$

Therefore, the first expression is proportional to the second one:

$$\tilde{w}_t^{(i)} \propto w(x_{0:t}^{(i)}) = \frac{p(x_{0:t}^{(i)}|y_{1:t})}{\pi(x_{0:t}^{(i)}|y_{1:t})}$$

This expression can also be represented as:

$$\frac{p(x_{0:t}^{(i)}|y_{1:t})}{\pi(x_{0:t}^{(i)}|y_{1:t})} = \frac{p(x_{0:t}^{(i)}|y_{1:t})}{\pi(x_{0:t-1}^{(i)}|y_{1:t-1})\,\pi(x_t^{(i)}|x_{0:t-1}^{(i)}, y_{1:t-1})}$$

Focusing on the numerator, it can be reformulated as:

$$p(x_{0:t}^{(i)}|y_{1:t}) = \frac{p(y_{1:t}|x_{0:t}^{(i)})\,p(x_{0:t}^{(i)})}{\int p(y_{1:t}|x_{0:t}^{(i)})\,p(x_{0:t}^{(i)})\,dx_{0:t}^{(i)}}$$

This expression is proportional to the following:

$$\frac{p(y_{1:t}|x_{0:t}^{(i)})\,p(x_{0:t}^{(i)})}{\int p(y_{1:t}|x_{0:t}^{(i)})\,p(x_{0:t}^{(i)})\,dx_{0:t}^{(i)}} \propto p(y_{1:t}|x_{0:t}^{(i)})\,p(x_{0:t}^{(i)})$$

Now, keeping deriving from this just obtained expression:

$$p(y_{1:t}|x_{0:t}^{(i)})\,p(x_{0:t}^{(i)}) = p(y_t|x_{0:t}^{(i)})\,p(y_{1:t-1}|x_{0:t}^{(i)})\,p(x_{0:t}^{(i)})$$

The two right terms can be grouped up into:

$$p(y_t|x_{0:t}^{(i)})\,p(y_{1:t-1}|x_{0:t}^{(i)})\,p(x_{0:t}^{(i)}) = p(y_t|x_t^{(i)})\,p(x_{0:t}^{(i)}|y_{1:t-1})$$

Which, keeping expanding, provides:

$$\begin{aligned} p(y_t|x_t^{(i)})\,p(x_{0:t}^{(i)}|y_{1:t-1}) &= p(y_t|x_t^{(i)})\,p(x_t^{(i)}|y_{1:t-1}, x_{0:t-1}^{(i)})\,p(x_{0:t-1}^{(i)}|y_{1:t-1}) \\ &= p(y_t|x_t^{(i)})\,p(x_t^{(i)}|x_{t-1}^{(i)})\,p(x_{0:t-1}^{(i)}|y_{1:t-1}) \end{aligned}$$

And from here, substituting in the previous fractional expression, and applying the proportion of the beginning:

$$\tilde{w}_t^{(i)} \propto \frac{p(y_t|x_t^{(i)})\,p(x_t^{(i)}|x_{t-1}^{(i)})\,p(x_{0:t-1}^{(i)}|y_{1:t-1})}{\pi(x_{0:t-1}^{(i)}|y_{1:t-1})\,\pi(x_t^{(i)}|x_{0:t-1}^{(i)}, y_{1:t-1})}$$

$$\tilde{w}_t^{(i)} \propto \tilde{w}_{t-1}^{(i)} \frac{p(y_t|x_t^{(i)})\,p(x_t^{(i)}|x_{t-1}^{(i)})}{\pi(x_t^{(i)}|x_{0:t-1}^{(i)}, y_{1:t-1})}$$

Task 4.2: Coding Exercise

Task (a):

Code can be found in the appendix.

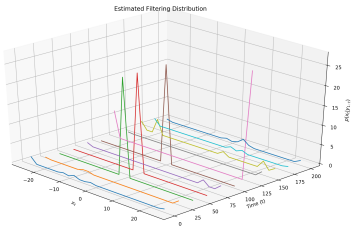

Figure 8: Bootstrap filter with 10 particles.

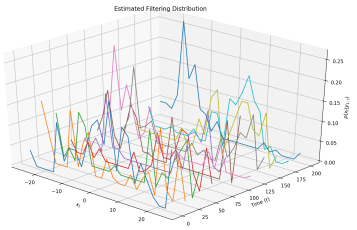

Figure 9: Bootstrap filter with 100 particles.

As it can be seen, the obtained distributions when using 10 particles are not multimodal. Only one peak is shown in most of the distributions. Nevertheless, when using 100 particles, it can be seen how in all different distributions several peaks are shown. Therefore, with 100 particles, it is possible to observe multimodality.

Task (b):

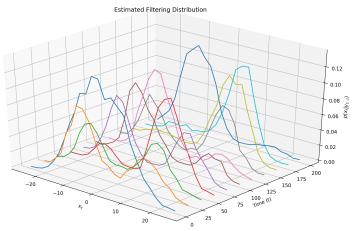

Figure 10: Bootstrap filter with 10000 particles.

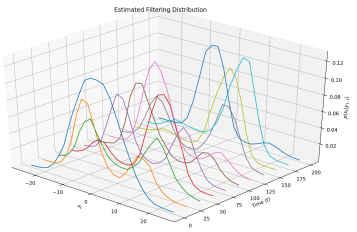

Figure 11: Bootstrap filter with 100000 particles.

As it is shown in the above plots, when using a big enough number of particles, the obtained distributions practically do not change. Therefore, for an $N = 10000$, convergence is obtained in the inferred distributions.

Metropolis-Hastings

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from tqdm import tqdm


def gaussian_mixture(a1, a2, mu1, mu2, sigma1, sigma2, x):  # Generates the Gaussian Mixture
    return a1 * stats.norm.pdf(x, mu1, sigma1) + a2 * stats.norm.pdf(x, mu2, sigma2)


def metropolis_hastings(iterations, a1, a2, mu1, mu2, sigma1, sigma2, proposal_variance):
    x_i = 0  # Initialize sample as zero
    accepted = []  # List of accepted samples
    for _ in tqdm(range(iterations)):
        x_star = stats.norm(x_i, proposal_variance).rvs(1)  # Generate sample from proposal distribution
        p_x_star = gaussian_mixture(a1, a2, mu1, mu2, sigma1, sigma2, x_star)  # Generate numerator p(x_star)
        p_x_i = gaussian_mixture(a1, a2, mu1, mu2, sigma1, sigma2, x_i)  # Generate denominator p(x_i)
        q_x_star = stats.norm.pdf(x_star, loc=x_i, scale=proposal_variance)  # Likelihood x_star
        q_x_i = stats.norm.pdf(x_i, loc=x_star, scale=proposal_variance)  # Likelihood x_i
        u = np.random.uniform(0, 1)  # Generate random sample from uniform distribution
        if u < ((p_x_star * q_x_i) / (p_x_i * q_x_star)):  # Check if the sample is accepted
            x_i = x_star  # Update state x as state x_star(i)
            accepted.append(x_i)  # Add accepted sample to the list
        else:
            accepted.append(x_i)  # Not sample update. Append same sample to list
    return np.array(accepted)


def main():
    a1, a2 = 0.5, 0.5  # Factors of the Gaussian Mixture
    mu1, mu2 = 0, 3  # μ of the Gaussian Mixture
    sigma1, sigma2 = 1, 0.5  # sigma of the Gaussian Mixture
    iterations = 10000  # Total number of iterations
    proposal_variance = 100  # Variance of the proposal distribution
    accepted_samples = metropolis_hastings(iterations, a1, a2, mu1, mu2, sigma1, sigma2, proposal_variance)
    fig, ax = plt.subplots()
    sns.distplot(accepted_samples)  # Plotting predicted distribution
    # Generating real distribution
    real_distribution = gaussian_mixture(a1, a2, mu1, mu2, sigma1, sigma2,
                                         np.arange(min(accepted_samples), max(accepted_samples), 1 / iterations))
    sns.lineplot(np.arange(min(accepted_samples), max(accepted_samples), 1 / iterations),
                 np.array(real_distribution), ax=ax, color='r')  # Plotting real distribution
    plt.show()
    plt.plot(accepted_samples)  # Plotting random walk
    plt.xlabel('Iteration')
    plt.ylabel('Value of x')
    plt.show()


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

Bootstrap Filter

```python
import matplotlib.pyplot as plt
import numpy as np
import random
import seaborn as sns
from scipy.stats import norm
from tqdm import tqdm
from mpl_toolkits.mplot3d import Axes3D

PLT_LAST = True  # Plotting the last obtained distribution individually


def generate_x(x_old, t, sigma_v, n):
    return (1 / 2) * x_old + 25 * (x_old / (1 + pow(x_old, 2))) + \
           8 * np.cos(1.2 * t) + np.random.normal(0, sigma_v, n)


def generate_y(n, time, sigma_x, sigma_v, sigma_w):
    x = np.zeros((time, n))
    y = np.zeros((time, n))
    # Generating x and y at t=0
    x[0] = np.random.normal(0, sigma_x, n)
    y[0] = (pow(x[0], 2) / 20) + np.random.normal(0, sigma_w, n)
    # Generating remaining x and y
    for t in range(1, time):
        x[t] = generate_x(x[t - 1], t, sigma_v, n)
        y[t] = (pow(x[t], 2) / 20) + np.random.normal(0, sigma_w, n)
    return y


def calculate_weights(x, y, sigma):
    weights = generate_distribution(x, y, sigma)  # Generating weights
    weights /= np.sum(weights)  # Normalizing weights
    return weights


def generate_distribution(x, y, sigma_w):
    return norm.pdf(y, loc=(pow(x, 2) / 20), scale=sigma_w)


def bootstrap(n, time, sigma_x, sigma_v, sigma_w):
    y = generate_y(n, time, sigma_x, sigma_v, sigma_w)  # Obtain y
    x = np.zeros((time, n))  # Initialize x
    positions = np.array(range(0, n))  # Create array of particle positions
    print("Executing bootstrap algorithm...")
    x[0] = np.random.normal(0, sigma_x, n)  # Sample the first x
    weights = calculate_weights(x[0], y[0], sigma_w)
    selection = random.choices(positions, weights=weights, k=n)  # Select randomly
    according to the weights
    x[0] = x[0, selection]  # Update x at time=0 with the selected samples
    for t in tqdm(range(1, time)):
        x[t] = generate_x(x[t - 1], t, sigma_v, n)  # Sample x given old x
        weights = calculate_weights(x[t], y[t], sigma_w)
        selection = random.choices(positions, weights=weights, k=n)
        x[t] = x[t, selection]  # Update x at time=t with the selected samples

    return x


def main():
    n_plots = 10  # Number of pdf drawn
    n = 25  # Number of bins/cells in grid
    time = 200  # Number of time stamps
    particles = 10000  # Number of particles generated
    sigma_x, sigma_v, sigma_w = np.sqrt(10), np.sqrt(10), np.sqrt(1)  # sigmas of the
    normal distributions
    x = bootstrap(particles, time, sigma_x, sigma_v, sigma_w)
    # Plotting
    x_grid = np.linspace(-25, 25, n)  # Grid for plotting x
    timestamps = np.arange(0, time, time / n_plots)  # Number of plots (equal distance
```

```
    in between)
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    for i in range(len(timestamps)):
        t = timestamps[i] * np.ones(n)  # Generating the time array
        x_aux = x[int(timestamps[i]), :]  # Selecting the proper data for plotting
        density = np.histogram(x_aux, bins=n, density=True)[0]  # Getting the density
    from the data
        ax.plot(x_grid, t, density)  # Plotting the pdf
    # Last position calculated
    t = (time - 1) * np.ones(n)
    x_aux = x[(time - 1), :]
    density = np.histogram(x_aux, bins=n, density=True)[0]
    ax.plot(x_grid, t, density)
    ax.set_xlabel('$x_{t}$')
    ax.set_ylabel('Time (t)')
    ax.set_zlabel('$p(x_{t}|y_{1:t})$')
    plt.title("Estimated Filtering Distribution")
    plt.show()
    if PLT_LAST:  # Plot the last distribution
        plt.figure()
        sns.distplot(x[time - 1, :])
        plt.show()


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

References

[1] Analytica. Importance weights. `http://wiki.analytica.com/Importance_weights`. Accessed: 2020-02-16.

[2] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[3] M. L. Rizzo and G. J. Székely. Energy distance. *WIREs Comput. Stat.*, 8(1):27–38, Jan. 2016.

[4] C. Robert and G. Casella. *Monte Carlo statistical methods*. Springer Science & Business Media, 2013.

[5] I. Yildirim. Bayesian inference: Metropolis-hastings sampling. *Dept. of Brain and Cognitive Sciences, Univ. of Rochester, Rochester, NY*, 2012.