AI701 Assignment 2

Sultan Rizky Hikmawan Madjid (20214196)

Due: October 27, 2021

Problem 1 (20 pts)

(a) For an exponential random variable $X \sim \text{Exp}(\lambda)$ with parameter $\lambda > 0$ and probability density function (PDF)

$$f_X(x) = \lambda e^{-\lambda x} \mathbb{1}[x \geq 0]\,,$$

Using the function `npr.read` to generate sample u from a uniform distribution $u \sim U(0,1)$, it is possible to employ Inverse CDF method to sample X from u. Defining $F_X(x)$ to be the CDF of X, we have that

$$\begin{aligned} F_X(x) &= \int_{-\infty}^{x} f_X(x')\mathrm{d}x' = \lambda \int_0^x e^{-\lambda x'}\mathrm{d}x' \\ &= \left[-e^{-\lambda x'}\right]_0^x = 1 - e^{-\lambda x}. \end{aligned}$$

From the above expressions, we have that the quantile of X can be expressed as

$$F_X^{-1}(u) = -\frac{\log(1-u)}{\lambda}.$$

Using the fact that $u \overset{d}{=} 1-u$, we have that

$$u \sim U(0,1), \quad x := -\frac{\log u}{\lambda} \overset{d}{=} \text{Exp}(\lambda)$$

The following snippet of code tries to sample X using samples generated from u.

```
import numpy as np
import numpy.random as npr

# Draw n samples from an exponential distribution with parameter lamb.
def rand_exp(lamb, n):
        u = npr.rand(n)
        # Using inverse CDF method
        x = -np.log(u)/lamb
        return x
```

(b) Given an exponential random variable $X \sim \text{Exp}(\lambda)$ with parameter $\lambda > 0$ and probability density function (PDF)

$$f_X(x) = \lambda e^{-\lambda x} \mathbb{1}[x \geq 0]\,,$$

We have that $E[X]$, which denotes the mean of X, expressed as

$$\begin{aligned} E[X] &= \int_{-\infty}^{\infty} x\lambda e^{-\lambda x}\mathbb{1}[x \geq 0]\mathrm{d}x \\ &= \frac{1}{\lambda}\int_0^{\infty} y e^{-y}\mathrm{d}y \qquad \text{(Perform substitution } y = \lambda x\text{)} \\ &= \frac{1}{\lambda}\left[-ye^{-y} - e^{-y}\right]_0^{\infty} \\ &= \frac{1}{\lambda} \end{aligned}$$

Similarly, $\mathrm{Var}[X]$, which represents the variance of X, can be expressed as

$$\begin{aligned}
\mathrm{Var}[X] &= E\left[X^2\right] - (E[X])^2 \\
&= \int_{-\infty}^{\infty} x^2 \lambda e^{-\lambda x} \mathbb{1}[x \geq 0] \mathrm{d}x - \frac{1}{\lambda^2} \\
&= \frac{1}{\lambda^2} \int_0^{\infty} y^2 e^{-y} \mathrm{d}y - \frac{1}{\lambda^2} && \text{(Perform substitution } y = \lambda x\text{)} \\
&= \frac{1}{\lambda^2} \left(\left[e^{-y} \left(y^2 + 2y + 2 \right) \right]_0^{\infty} - 1 \right) \\
&= \frac{1}{\lambda^2}
\end{aligned}$$

(c) Using the result obtained in b, the following code shows the correctness for `rand_exp` we implemented in a.

```
# compute the mean of an exponential distribution with parameter lamb.
def exp_mean(lamb):
        return 1/lamb

# compute the variance of an exponential distribution with parameter lamb.
def exp_var(lamb):
        return 1/(lamb)**2

lamb = 1.5
n = 100000
x = rand_exp(lamb, n)

print(f'true mean {exp_mean(lamb)}, empirical mean {x.mean()}')
print(f'true variance {exp_var(lamb)}, empirical variance {x.var()}')
```

Here are the results we obtained while executing the code.

```
true mean 0.6666666666666666, empirical mean 0.6656982498031476
true variance 0.4444444444444444, empirical variance 0.44331074868144876
```

The negligible difference between true and empirical mean and variance here shows that the implementation at a works properly.

(d) Now, considering a gamma random variable $Y_1 \sim \mathrm{Gamma}(a_1, b)$ with shape parameter $a_1 > 0$ and rate parameter $b > 0$ with PDF

$$f_{Y_1}(y) = \frac{b^{a_1} y^{a_1 - 1} e^{-by}}{\Gamma(a_1)} \mathbb{1}[y \geq 0]$$

Taking another gamma random variable $Y_2 \sim \mathrm{Gamma}(a_2, b)$ with shape parameter $a_2 > 0$ and rate parameter $b > 0$, assuming that Y_1 and Y_2 are independent, we have that for a random variable $Y = Y_1 + Y_2$, its PDF $f_Y(y)$ can be expressed as

$$\begin{aligned}
f_Y(y) &= \int_{-\infty}^{\infty} f_{Y_1}(t) f_{Y_2}(y - t) \mathrm{d}t \\
&= \frac{b^{a_1 + a_2} e^{-by}}{\Gamma(a_1)\Gamma(a_2)} \int_0^y t^{a_1 - 1} (y - t)^{a_2 - 1} \mathrm{d}t \\
&= \frac{b^{a_1 + a_2} e^{-by}}{\Gamma(a_1)\Gamma(a_2)} y^{a_1 + a_2 - 1} \int_0^1 z^{a_1 - 1} (1 - z)^{a_2 - 1} \mathrm{d}z && \text{(Perform substitution } t = yz\text{)} \\
&= \frac{b^{a_1 + a_2} y^{a_1 + a_2 - 1} e^{-by}}{\Gamma(a_1)\Gamma(a_2)} \frac{\Gamma(a_1)\Gamma(a_2)}{\Gamma(a_1 + a_2)} = \frac{b^{a_1 + a_2} y^{a_1 + a_2 - 1} e^{-by}}{\Gamma(a_1 + a_2)}
\end{aligned}$$

Which corresponds to a gamma distribution $Y \sim \mathrm{Gamma}(a_1 + a_2, b)$. □

(e) In this section, we are going to prove that a gamma distributed random variable $Y \sim \text{Gamma}(m, 1)$ with $m \in \mathbb{N}$ can be obtained from summation of m exponential random variables $X \sim \text{Exp}(1)$, which can be simply referred to as $X \sim \text{Gamma}(1, 1)$.
We learned from d that summation of two independent gamma random variables $Y_1 \sim \text{Gamma}(a_1, b)$ and $Y_2 \sim \text{Gamma}(a_2, b)$ results in a gamma random variable $Y = Y_1 + Y_2$ where $Y \sim \text{Gamma}(a_1 + a_2, b)$. Since Exp(1) corresponds to the same distribution as Gamma(1, 1), by fixing $a_1 = a_2 = 1$, it will result in a random variable Y with $Y \sim \text{Gamma}(2, 1)$. Summing up Y with X for once more (by making Y and X independent towards each other) will result in another gamma distribution Y' with $Y' \sim \text{Gamma}(3, 1)$.
Using this logic, if we sum up m samples generated from m independent exponential random variables that follow the exponential distribution Exp(1) or $X \sim \text{Gamma}(1, 1)$, it will ultimately result in a sample that follows the gamma distribution $Y \sim \text{Gamma}(m, 1)$.

(f) Using the procedure explained in e, the following code tries to draw samples from $Y \sim \text{Gamma}(m, 1)$ and verify its correctness.

```
# Draw n samples from a gamma random variable with shape m (natural number)
# and rate 1.
def rand_gamma(m, n):
        f_y = 0
        # Using the facts derived at (e)
        for i in range(m):
                f_y += rand_exp(1, n)
        return f_y

def gamma_mean(m):
        return m

def gamma_var(m):
        return m

n = 100000
m = 6
x = rand_gamma(m, n)

print(f'true mean {gamma_mean(m)}, empirical mean {x.mean()}')
print(f'true variance {gamma_var(m)}, empirical variance {x.var()}')
```

Here are the results we obtained while executing the code.

```
true mean 6, empirical mean 6.000319884866008
true variance 6, empirical variance 5.945326011390203
```

The negligible difference between true and empirical mean and variance here shows that the idea proposed at e works properly.

Problem 2 (30 pts)
For a random variable X whose unnormalized PDF is given as follows:

$$\tilde{f}_X(x) = x^2 \sin(2\pi x)\mathbb{1}[-2 \le x \le 2], \quad f_X(x) = \frac{\tilde{f}_X(x)}{Z}, \quad Z := \int_{-2}^{2} \tilde{f}_x(x)\mathrm{d}x,$$

Using a proposal distribution with density function $q(x)$ given by

$$q(x) = \frac{3}{16}x^2\mathbb{1}[-2 \le x \le 2]$$

and $M = 16/3$,

(a) To sample from $q(x)$ using `npr.rand` (drawing samples from $u \sim U(0,1)$), we simply employ the Inverse CDF method to sample proposal distribution with density $q(x)$ from u. Taking $Q(x)$ to be the CDF of the proposal distribution, we have that for $x \in [-2, 2]$,

$$\begin{aligned} Q(x) &= \int_{-\infty}^{x} q(x')\mathrm{d}x \\ &= \frac{3}{16}\int_{-2}^{x} x'^2\mathrm{d}x' \\ &= \frac{1}{16}\left[x'^3\right]_{-2}^{x} = \frac{1}{16}\left(x^3 + 8\right) \end{aligned}$$

From the above expressions, we have thet the quantile of proposal distribution can be expressed as

$$Q^{-1}(u) = \sqrt[3]{16u - 8}$$

Therefore, we have that

$$u \sim U(0,1), \quad q(x) := \sqrt[3]{16u - 8}$$

We also have that $E_{q(x)}[X]$, which denotes the mean of $q(x)$, expressed as

$$\begin{aligned} E_{q(x)}[X] &= \int_{-\infty}^{\infty} xq(x)\mathrm{d}x \\ &= \frac{3}{16}\int_{-2}^{2} x^3\mathrm{d}x \\ &= \frac{3}{64}\left[x^4\right]_{-2}^{2} = 0 \end{aligned}$$

Similarly, we have that $\mathrm{Var}_{q(x)}[X]$, which denotes the variance of $q(x)$, expressed as

$$\begin{aligned} \mathrm{Var}_{q(x)}[X] &= E_{q(x)}[X^2] - \left(E_{q(x)}[X]\right)^2 \\ &= \int_{-\infty}^{\infty} x^2q(x)\mathrm{d}x \\ &= \frac{3}{16}\int_{-2}^{2} x^4\mathrm{d}x \\ &= \frac{3}{80}\left[x^5\right]_{-2}^{2} = \frac{12}{5} \end{aligned}$$

(b) Using the intuitions and calculations performed in a, the following code tries to draw samples from $q(x)$ and verify its correctness.

```
# draw n samples from the distribution with PDF q(x).
def rand_q(n):
        # Initialize the uniform distribution u_0 ~ U(0,1)
        u_0 = npr.rand(n)
        # Using Inverse CDF Method
        q = np.cbrt(16*u_0 - 8)
        return q
```

```
q_true_mean = 0
q_true_var = 2.4


n = 100000
x = rand_q(n)

print(f'true mean {q_true_mean}, empirical mean {x.mean()}')
print(f'true variance {q_true_var}, empirical variance {x.var()}')
```

Here are the results we obtained while executing the code.

```
true mean 0, empirical mean 0.005791076221952388
true variance 2.4, empirical variance 2.4015741588997255
```

The negligible difference between true and empirical mean and variance here shows that the idea proposed at a works properly.

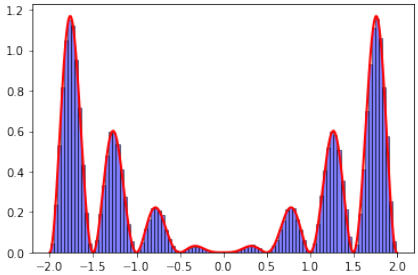
(c) To sample from $f_X(x)$ with rejection sampling using $q(x)$ as a proposal with $M = 16/3$, the function `rand_f_rejection` is implemented, with the correctness of the function is done by drawing histograms of samples and comparing empirical means and variances to numerically computed means and expectations done by the following code snippet.

```
# draw a sample from f_X with rejection sampling.
def rand_f_rejection():
        while 1:
                x = rand_q(1)
                u = npr.rand(1)
                crit = f_tilde(x)/(M*q(x))
                if (u < crit):
                        return x

# draw samples
n = 100000
x = np.array([rand_f_rejection() for _ in range(n)])

# verification by drawing empirical distribution

plt.hist(x, bins=100, density=True, facecolor='b', edgecolor='k', alpha=0.5);

# numerical integration for the normalization constant
Z = quad(f_tilde, -2, 2)[0]
tx = np.linspace(-2, 2, 1000)
plt.plot(tx, f_tilde(tx)/Z, '-r', linewidth=2.0)

# verification by comparing mean and variances
numerical_mean = quad(lambda x: x*f_tilde(x)/Z, -2, 2)[0]
numerical_var = quad(lambda x: x**2*f_tilde(x)/Z, -2, 2)[0] - numerical_mean**2

print(f'numerical mean {numerical_mean}, empirical mean {x.mean()}')
print(f'numerical variance {numerical_var}, empirical variance {x.var()}')
```

Here are the results we obtained while executing the code.

```
numerical mean 0.0, empirical mean 0.0019016832297448182
numerical variance 2.3470250033370554, empirical variance 2.3435128105756378
```

The negligible difference between true and empirical mean and variance here along with the histogram of samples that fits below the $f_X(x)$ shows that our algorithm works properly.



(d) Now, considering the expectation X

$$E\left[g(X)\right] = \int_{-2}^{2} g(x) f_X(x) \mathrm{d}x = \int_{-2}^{2} g(x) \frac{\hat{f}_X(x)}{Z} \mathrm{d}x$$

Using the self-normalizing importance sampling (SNIS) technique with defined $q(x)$ to estimate this expectation, the following code implements this sampling technique followed by checking the correctness for our implementation.

```
# compute the approximation for the expectation E[g(X)] using n number of samples.
# g: target function to compute expectation
# n: number of samples being used
def SNIS(g, n):
    # Draw samples from q(x)
        x = rand_q(n)
        # Calculate \omega and perform estimation
        omega = f_tilde(x)/q(x)
        mu_est = (omega*g(x)/(omega.sum())).sum()
        return mu_est


# verify your implementation using numerical integration
g = lambda x: np.log(np.abs(x) + 1.0)
n = 100000
snis_expec = SNIS(g, n)

numerical_expec = quad(lambda x: g(x)*f_tilde(x)/Z, -2, 2)[0]

print(f'Expectation via numerical integration {numerical_expec},
expectation via SNIS {snis_expec}')
```

Here are the results we obtained while executing the code.

```
Expectation via numerical integration 0.8974560004907551,
expectation via SNIS 0.8969511403159071
```

The negligible difference between expectation obtained through numerical integration and SNIS shows the correctness of our algorithm with respect to the true value.

Problem 3 (20 pts)
For a d-dimensional categorical random variable X with Probability Mass Function (PMF)

$$f_X(x) = \prod_{j=1}^{d} \pi_j^{\mathbb{1}[x=j]}$$

The following code tries to draw samples from said distribution via `npr.rand` and check the correctness for our implementation

```
# draw a sample from a categorical distribution with parameter pi
def rand_cat(pi):
        # Compute cumulative probability theta_hat
        theta_hat = np.array([pi[:i].sum() for i in range(1,pi.size+1)])
        # Draw u ~ Unif(0,1)
        u = npr.rand(1)
        x = 0
        while u > theta_hat[x]:
                x += 1
        return x

# verify the code using LLN
d = 5
pi = npr.rand(d)
pi = pi / pi.sum()

n = 100000
x = np.zeros(d)
for i in range(n):
        x[rand_cat(pi)] += 1

print(f'empirical mean {x/x.sum()}')
print(f'true mean {pi}')
```

Here are the results we obtained while executing the code.

```
empirical mean [0.16086 0.00435 0.10372 0.3713  0.35977]
true mean [0.16184142 0.00432721 0.10369096 0.37076513 0.35937528]
```

The negligible difference between empirical mean and true mean shows the correctness of our algorithm with respect to the true value.

Problem 4 (30 pts)
For a bivariate random variable $X = (X_1, X_2)$ with the following unnormalized PDF:

$$\hat{f}_X(x_1, x_2) = \exp\left(-\left(x_1 - \frac{x_2^2}{4}\right)^2 - \frac{x_2^2}{4}\right)$$

Using a sample random walk Gaussian distribution $q(x'|x)$ as our proposal

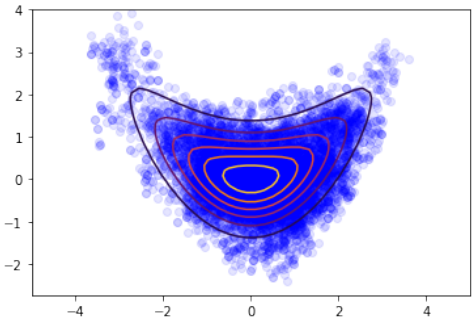
$$q(x'|x) = \mathcal{N}(x'; x, \sigma^2 I)$$

where we fix $\sigma = 0.2$, we are going to implement a Metropolis-Hastings algorithm based Markov-Chain Monte-Carlo sampler for the target distribution $\hat{f}_X(x_1, x_2)$. To do so, first, we draw the contour plot of the density. Next, we implement the algorithm to run the random walk Metropolis-Hastings, and finally, we verify the correctness of our implementation by visualizing samples provided. The code is as follows:

```
# x: n times 2 matrix or 2 dimensional vector
def log_f_tilde(x):
        if x.ndim == 2:
                x1, x2 = x[:,0], x[:,1]
        else:
                x1, x2 = x

        return -(x2 - x1**2/4)**2 - x1**2/4

def plot_density(alpha=1.0):
        nx, ny = 50, 50
        x = np.linspace(-5, 5, nx)
        y = np.linspace(-2, 4, ny)
        xx, yy = np.meshgrid(x, y)
        z = np.exp(log_f_tilde(np.concatenate([xx.reshape((-1, 1)),
        yy.reshape((-1, 1))], -1)))
        plt.contour(x, y, z.reshape((nx, ny)), cmap='inferno', alpha=alpha)

plot_density()

# propose a sample from the proposal distribution q(x' | x) = N(x' ; x, sigma^2*I)
def q_MH(x, sigma):
        # Sample epsilon_1, epsilon_2 ~ N(0,1)
        epsilon = npr.normal(0, 1, x.size)
        # Sample x_acc ~ N(x, sigma^2*I)
        x_acc = x + sigma*epsilon
        return x_acc

# run a random-walk Metropolis Hastings
# x0: initial sample
# num_samples: number of samples to collect
# sigma: variance for the proposal distribution
def RWMH(x0, num_samples, sigma):
        x = np.expand_dims(x0, axis=0)
        for i in range(1, num_samples):
                x_i = x[-1,:]
                # Generate x_acc from q_MH
                x_acc = q_MH(x_i, sigma)
                # Compute Accptance Probability
                Acc_Prob = min(1, np.exp(log_f_tilde(x_acc) - log_f_tilde(x_i)))
                # Draw a uniform random number u ~ U(0,1)
                u = npr.rand(1)
                # Accept if u <= Acc_Prob, Reject otherwise
                if u <= Acc_Prob:
```

```
                x = np.append(x, np.expand_dims(x_acc, axis=0), axis=0)
            else:
                x = np.append(x, np.expand_dims(x_i, axis=0), axis=0)
        return x

# randomly initialize a chain
x0 = 0.01*npr.randn(2)
# collect 10000 samples from RWMH
x = RWMH(x0, 10000, 0.2)

# visualize the samples
plot_density(alpha=1.0)
plt.scatter(x[:,0], x[:,1], alpha=0.1, color='b')
```

Here are the results we obtained while executing the code. Since the contained samples are distributed



tightly around the contour plot, it can be verified that our algorithm works properly.