FIM 548-001 HW-1

Yi-Siou Feng, Raiden Han, Tingyu Lei

January 24, 2022

Perface

All codes in this report are Python codes. To implement these methods, we need to import the NumPy package initially.

```
import numpy as np
np.random.seed(124)
```

Problem 1

For $0 \leq x \leq 1$, the cumulative distribution function is

$$F(x) = \int_0^x 2(1-u)\,du = 2x - x^2, \quad 0 \leq x \leq 1.$$

To use the inverse transform method, we need the inverse function of the cdf. Let us assume $F(x) = u$, and solve the equation inversely.

$$F(x) = 2x - x^2 = u,$$
$$x = 1 \pm \sqrt{1-u}.$$

Because this equation applies to $0 \leq x \leq 1$ by definition, the inverse function is

$$F^{-1}(u) = 1 - \sqrt{1-u}, \quad 0 \leq u \leq 1.$$

Thus, we only need to generate random variable $U \sim Unif(0,1)$, and the corresponding $X = F^{-1}(U) = 1 - \sqrt{1-U}$ will conform to our pdf.

```
n_samples = int(1e7)
unif = np.random.uniform(0, 1, n_samples)
rt_dist = 1 - np.sqrt(1 - unif)
```

To cross-validate our results, we plot the histogram of the generated random variable and the actual probability density function, and the result is displayed in Figure 1. The plotting codes used here and below are shown in full in Appendix 1, and will not be repeated thereafter.

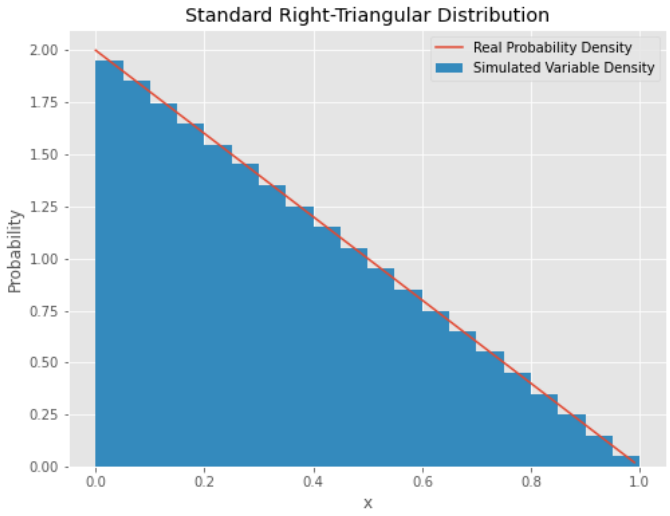


Figure 1: Standard Right-Triangular Distribution

Problem 2

Similarly, we calculate the cumulative distribution function

$$F(x) = \begin{cases} \int_{-\infty}^{x} \frac{1}{2}e^{x} = \frac{1}{2}e^{x}, & x < 0 \\ \int_{-\infty}^{0} \frac{1}{2}e^{x} + \int_{0}^{x} \frac{1}{2}e^{-x} = 1 - \frac{1}{2}e^{-x}, & x \geq 0. \end{cases}$$

Let $F(x) = u$, and solve the equations inversely.

$$u = \begin{cases} \frac{1}{2}e^{x}, & x < 0 \\ 1 - \frac{1}{2}e^{-x}, & x \geq 0, \end{cases}$$

$$F^{-1}(u) = x = \begin{cases} \log(2u), & 0 < u < \frac{1}{2} \\ -\log(2 - 2u), & u \geq \frac{1}{2}, \end{cases}$$

Again, we simply generate a random variable $U \sim Unif(0, 1)$ and calculate $X = F^{-1}(U)$, which conforms to the double exponential distribution. To improve the time efficiency, we simply use the generated uniform distribution from Problem 1. The comparison is shown in Figure 2.

```
doub_exp_dist = np.where(unif < 1 / 2, np.log(2 * unif), -np.log(2 - 2 * unif))
```

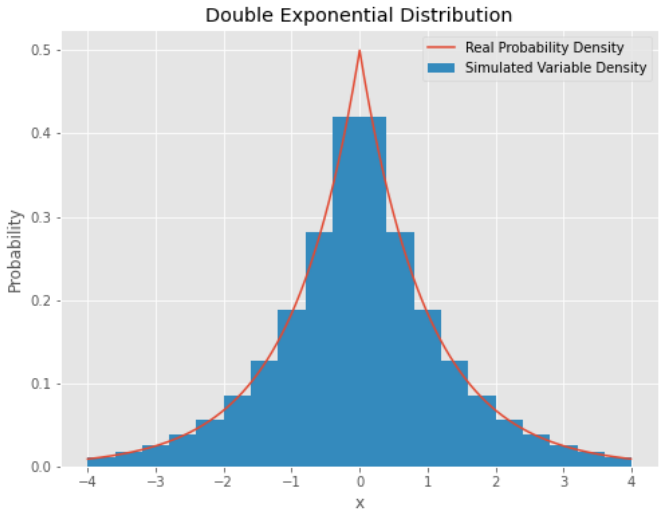


Figure 2: Double Exponential Distribution

Problem 3

Problem 3.1

First, let us find the constant $c(\lambda)$ so that $f(x)/g(x) \leq c(\lambda)$ for all x. Denote

$$h(x,\lambda) = \frac{f(x)}{g(x)} = \frac{x^3 e^{-x}/6}{\lambda e^{-\lambda x}} = \frac{x^3 e^{(\lambda-1)x}}{6\lambda}, \quad x \geq 0, \lambda > 0,$$

so

$$\frac{\partial}{\partial x} h(x,\lambda) = \frac{x^2 e^{(\lambda-1)x}[(\lambda-1)x+3]}{6\lambda}, \quad x \geq 0, \lambda > 0.$$

Since $x^2 > 0$, $e^{(\lambda-1)x} > 0$ and $6\lambda > 0$, it is obvious that for a certain $\lambda > 0$, $\frac{\partial}{\partial x} h(x,\lambda) = 0$ has at most one solution, which is

$$x = \frac{3}{1-\lambda}, \quad 0 < \lambda < 1.$$

If $0 < \lambda < 1$, $\lambda - 1 < 0$. $\frac{\partial}{\partial x} h(x,\lambda) > 0$ for $x \in [0, \frac{3}{1-\lambda})$ and $\frac{\partial}{\partial x} h(x,\lambda) < 0$ for $x \in (\frac{3}{1-\lambda}, \infty)$. Therefore, the maximum $h(x,\lambda)$ in terms of x is

$$\max_x h(x,\lambda) = \left.\frac{x^3 e^{(\lambda-1)x}}{6\lambda}\right|_{x=\frac{3}{1-\lambda}} = \frac{9}{2e^3\lambda(1-\lambda)^3}.$$

Then, if $\lambda \geq 1$, $\frac{\partial}{\partial x} h(x,\lambda) > 0$ for all $x \geq 0$, and the maximum $h(x)$ is undefined given

$$\lim_{x\to\infty} h(x,\lambda) = \lim_{x\to\infty} \frac{x^3 e^{(\lambda-1)x}}{6\lambda} = \infty.$$

As a result, if $0 < \lambda < 1$, there is a constant

$$c(\lambda) = \frac{9}{2e^3\lambda(1-\lambda)^3}$$

such that the AR method is feasible. Otherwise, $c(\lambda)$ does not exist.

Problem 3.2

To make the AR method most efficient, we need to find a λ such that $c(\lambda)$ is greater but close to 1 as much as possible. So we calculate

$$\frac{d}{d\lambda}c(\lambda) = \frac{9(4\lambda - 1)}{2e^3\lambda^2(1-\lambda)^4}, \quad 0 < \lambda < 1.$$

where the denominator is strictly great than 0. Let

$$\frac{d}{d\lambda}c(\lambda) = 0,$$

$$\lambda = \frac{1}{4}.$$

Needless proof, $\lambda = \frac{1}{4}$ minimizes $c(\lambda)$. And we also know that

$$c\left(\frac{1}{4}\right) = \frac{9}{2e^3\lambda(1-\lambda)^3}\bigg|_{\lambda=\frac{1}{4}} \approx 2.12 > 1,$$

which satisfies our requirement. To sum up, the optimal value is

$$\lambda^* = \frac{1}{4}.$$

Problem 4

To test the time efficiency of all the four algorithms, we use the 'timeit' magic command in the IPython environment. Also, we set a parameter at the start to control the total variable numbers. We transcribe the outputs in a comment format in these blocks, starting with '#>' as a hint for readability reasons. Besides, for the sake of time effectiveness and memory occupancy, we use the same codes with a smaller sample size $N_0 = 1,000,000$ to validate our algorithms in another .pyinb file. The results are shown in Figures 3-6, and the origin codes are attached in Appendix 2.

```
N = int(1e8)
```

Box-Muller method:

```
def BoxMuller(size):
    for i in range(size // 2):
        u1, u2 = np.random.uniform(size=2)
        param = np.sqrt(-2 * np.log(u1))
        x = param * np.cos(2 * np.pi * u2)
        y = param * np.sin(2 * np.pi * u2)

%timeit -r5 -n3 BoxMuller(N)
#> 7min 35s ±5.53 s per loop (mean ±std. dev. of 5 runs, 3 loops each)
```

Marsaglia-Bray polar method:

```
def Marsaglia(size):
    count = 0
    while count < size:
        v1, v2 = np.random.uniform(-1, 1, 2)
        s = v1 ** 2 + v2 ** 2
        if s <= 1:
            param = np.sqrt(-2 * np.log(s) / s)
            x = v1 * param
            y = v2 * param
            count += 2

%timeit -r5 -n3 Marsaglia(N)
#> 7min 20s ±1.6 s per loop (mean ±std. dev. of 5 runs, 3 loops each)
```

Beasley-Springer-Moro rational approximation:

```
def BSM_Rational(size):
    a = [2.50662823884, -18.61500062529, 41.39119773534, -25.44106049637]
    b = [-8.47351093090, 23.08336743743, -21.06224101826, 3.13082909833]
    c = [0.3374754822726147, 0.9761690190917186, 0.1607979714918209,
         0.0276438810333863, 0.0038405729373609, 0.0003951896511919,
         0.0000321767881768, 0.0000002888167364, 0.0000003960315187]
    for i in range(size):
        u = np.random.uniform()
        y = u - 0.5
        if np.abs(y) < 0.42:
            r = y * y
            x = y * (((a[3] * r + a[2]) * r + a[1]) * r + a[0]) / (
                       (((b[3] * r + b[2]) * r + b[1]) * r + b[0]) * r + 1)
        else:
            r = u
            if y > 0:
                r = 1 - u
            r = np.log(-np.log(r))
            x = c[0] + r * (c[1] + r * (c[2] + r * (c[3] + r * (c[4] + r * (
                   c[5] + r * (c[6] + r * (c[7] + r * c[8])))))))
            if y < 0:
                x = -x

%timeit -r5 -n3 BSM_Rational(N)
#> 6min 4s ±1.11 s per loop (mean ±std. dev. of 5 runs, 3 loops each)
```

Acceptance-Rejection method with a exponential distribution:

```
def AR_norm(size):
    count = 0
    while count < size:
        u1, u2 = np.random.uniform(size=2)
        y = -np.log(u2)
        if u1 <= np.exp(-(y - 1) ** 2 / 2):
            u3 = np.random.uniform()
            if u3 < 0.5:
                x = -y
```

```
            else:
                x = y
            count += 1

%timeit -r5 -n3 AR_norm(N)
#> 18min 41s ±3.22 s per loop (mean ±std. dev. of 5 runs, 3 loops each)
```

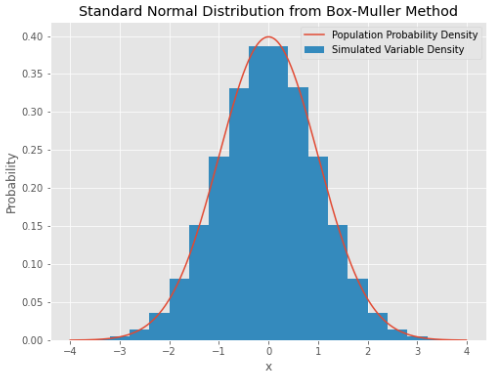


Figure 3: Box-Muller Method

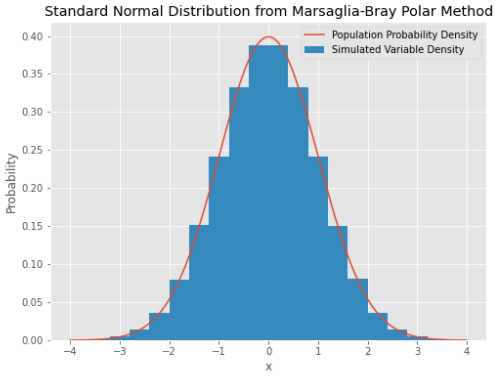


Figure 4: Marsaglia-Bray Polar Method

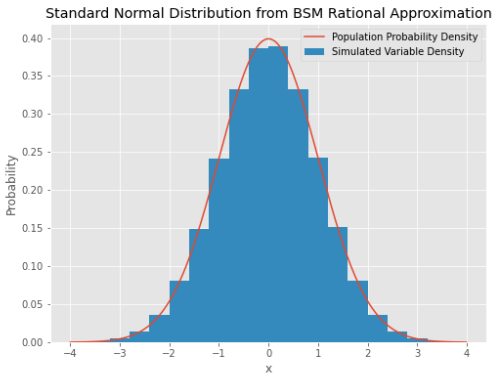


Figure 5: BSM Rational Approximation

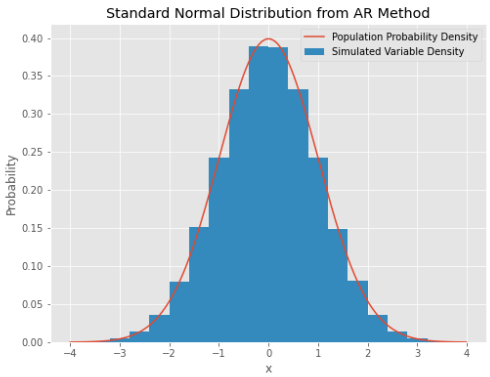


Figure 6: AR Method

From above, we can clearly learn that in the Python environment, Beasley-Springer-Moro rational approximation is the fastest algorithm to generate normal-distributed random variables. Box-Muller and Marsaglia-Bray polar methods' time efficiencies are on the same level, where the Marsaglia-Bray polar method is slightly faster than the other. AR method is the least efficient, taking around three times as long as Beasley-Springer-Moro rational approximation.

Problem 5

Run below codes, and we can find that both the expectation and the variance are approximately 1 from 100,000,000 simulations. For simplicity, we directly use the variable N defined in Problem 4.

```
cards = np.arange(1, 101)
order = np.arange(1, 101)
hits = []
for i in range(N):
    np.random.shuffle(cards)
    hits.append(np.sum(cards == order))

print('Expectation: {:.2f}'.format(np.mean(hits)))
print('Variance: {:.2f}'.format(np.var(hits)))
#> Expectation: 1.00
#> Variance: 1.00
```

Appendix 1 - Python Code

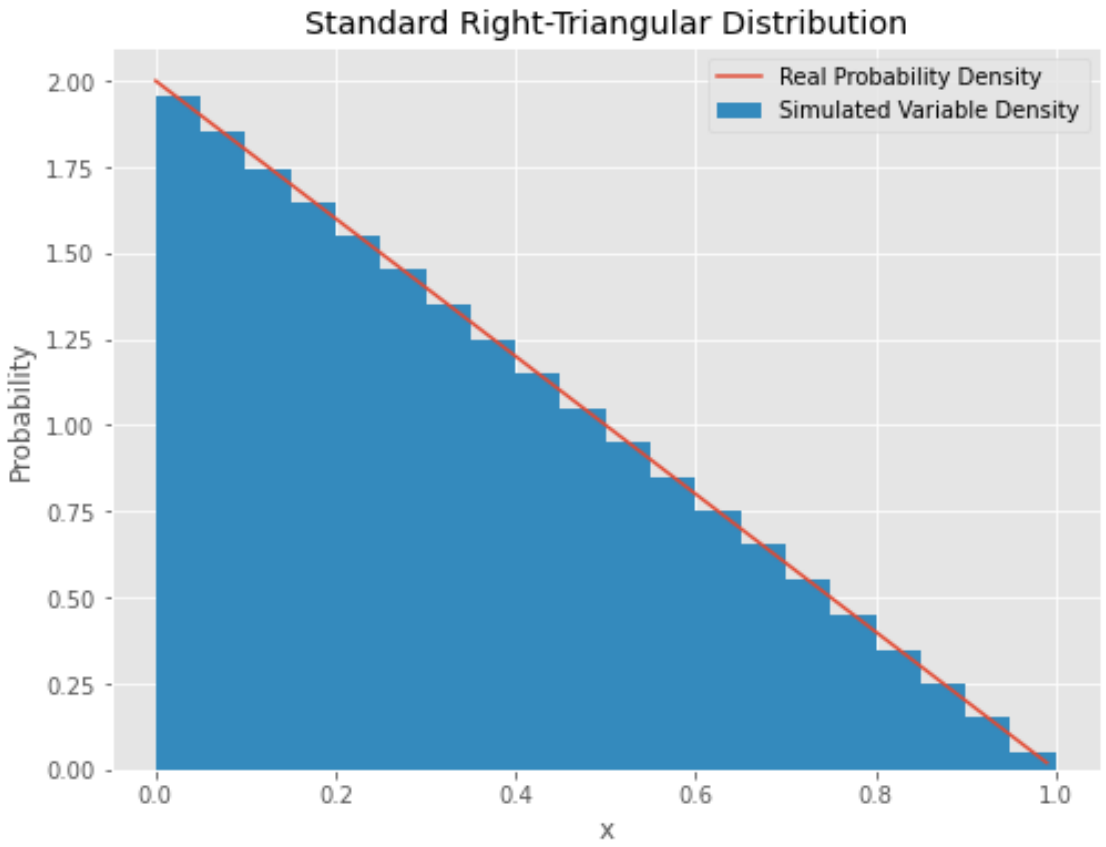
January 31, 2022

[1]:
```
# Load packages
import numpy as np
import matplotlib.pyplot as plt

# Specify the random seed and the plotting style
np.random.seed(124)
plt.style.use('ggplot')
```

1 Problem 1

$$f(x) = 2(1-x), 0 \leq x \leq 1$$
$$F(x) = 2x - x^2, 0 \leq x \leq 1$$
$$F^{-1}(x) = 1 - \sqrt{1-x}, 0 \leq x \leq 1$$

[2]:
```
# Generate the random variable using the inverse transform method
n_samples = int(1e7)
unif = np.random.uniform(0, 1, n_samples)
rt_dist = 1 - np.sqrt(1 - unif)

# Calculate the population density
x1 = np.arange(0, 1, .01)
y1 = 2 * (1 - x1)
# Plot the comparison figure
plt.figure(figsize=(8, 6))
plt.plot(x1, y1, label='Real Probability Density')
plt.hist(rt_dist, bins=20, density=True, label='Simulated Variable Density')
plt.xlabel('x')
plt.ylabel('Probability')
plt.legend()
plt.title('Standard Right-Triangular Distribution')
# plt.savefig('standard_right-triangular_distribution.png')
plt.show()
```



2 Problem 2

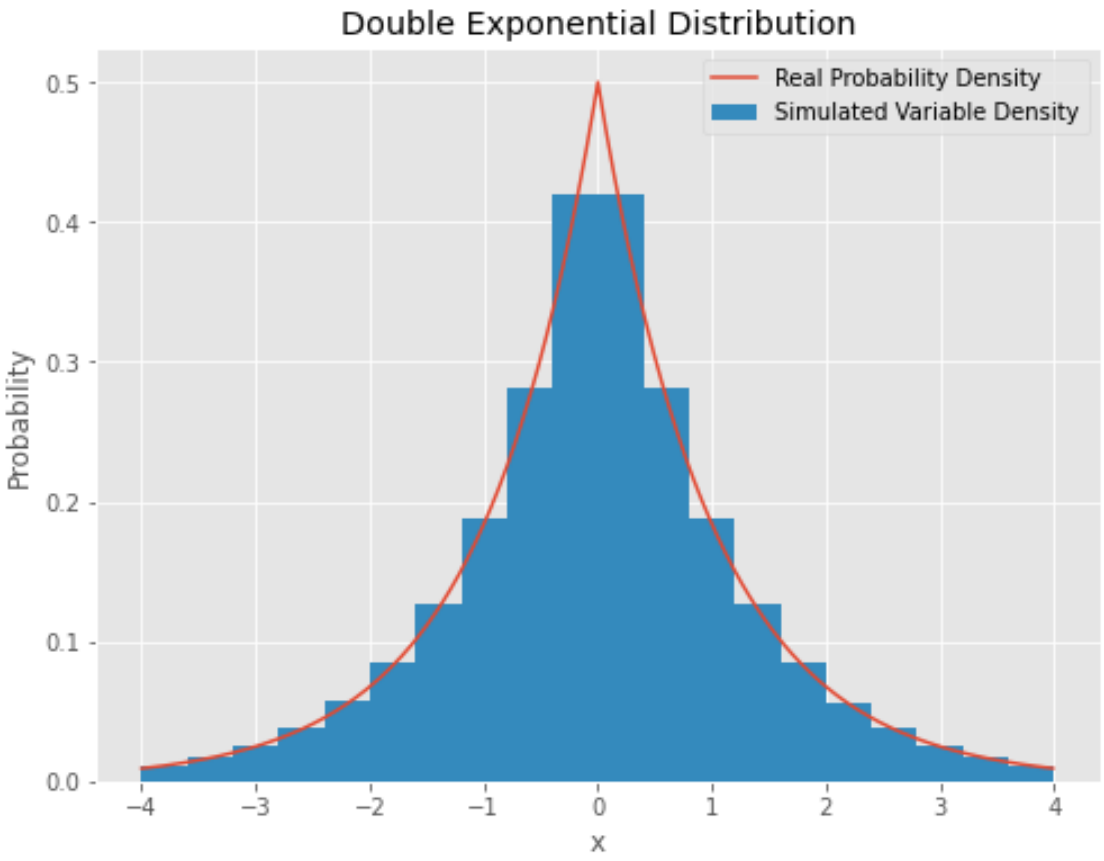
$$f(x) = \frac{1}{2}\mathbf{1}_{(-\infty,0)}(x)e^{x} + \frac{1}{2}\mathbf{1}_{[0,\infty)}(x)e^{-x}$$

$$F(x) = \begin{cases} \frac{1}{2}e^{x}, & x < 0 \\ 1 - \frac{1}{2}e^{-x}, & x \geq 0 \end{cases}$$

$$F^{-1}(x) = \begin{cases} \log(2x), & 0 \leq x < \frac{1}{2} \\ -\log(2 - 2x), & \frac{1}{2} \leq x \leq 1 \end{cases}$$

[3]:
```
# Generate the random variable using the inverse transform method
doub_exp_dist = np.where(unif < 1 / 2, np.log(2 * unif), -np.log(2 - 2 * unif))

# Calculate the population density
x2 = np.arange(-4, 4, .01)
y2 = np.where(x2 < 0, np.exp(x2) / 2, np.exp(-x2) / 2)
plt.figure(figsize=(8, 6))
plt.plot(x2, y2, label='Real Probability Density')
plt.hist(doub_exp_dist, bins=20, density=True, range=(-4, 4),
```

```
         label='Simulated Variable Density')
plt.xlabel('x')
plt.ylabel('Probability')
plt.legend()
plt.title('Double Exponential Distribution')
# plt.savefig('double_exponential_distribution.png')
plt.show()
```



3 Problem 4

[4]:
```
# Set the sample size
N = int(1e8)
```

[5]:
```
# Test the Box-Muller method
def BoxMuller(size):
    for i in range(size // 2):
        u1, u2 = np.random.uniform(size=2)
        param = np.sqrt(-2 * np.log(u1))
        x = param * np.cos(2 * np.pi * u2)
```

```
        y = param * np.sin(2 * np.pi * u2)

%timeit -r5 -n3 BoxMuller(N)
```

7min 35s ± 5.53 s per loop (mean ± std. dev. of 5 runs, 3 loops each)

[6]:
```
# Test the Marsaglia-Bray polar method
def Marsaglia(size):
    count = 0
    while count < size:
        v1, v2 = np.random.uniform(-1, 1, 2)
        s = v1 ** 2 + v2 ** 2
        if s <= 1:
            param = np.sqrt(-2 * np.log(s) / s)
            x = v1 * param
            y = v2 * param
            count += 2

%timeit -r5 -n3 Marsaglia(N)
```

7min 20s ± 1.6 s per loop (mean ± std. dev. of 5 runs, 3 loops each)

[7]:
```
# Test the Beasley-Springer-Moro rational approximation
def BSM_Rational(size):
    a = [2.50662823884, -18.61500062529, 41.39119773534, -25.44106049637]
    b = [-8.47351093090, 23.08336743743, -21.06224101826, 3.13082909833]
    c = [0.3374754822726147, 0.9761690190917186, 0.1607979714918209,
         0.0276438810333863, 0.0038405729373609, 0.0003951896511919,
         0.0000321767881768, 0.0000002888167364, 0.0000003960315187]
    for i in range(size):
        u = np.random.uniform()
        y = u - 0.5
        if np.abs(y) < 0.42:
            r = y * y
            x = y * (((a[3] * r + a[2]) * r + a[1]) * r + a[0]) / (
                        (((b[3] * r + b[2]) * r + b[1]) * r + b[0]) * r + 1)
        else:
            r = u
            if y > 0:
                r = 1 - u
            r = np.log(-np.log(r))
            x = c[0] + r * (c[1] + r * (c[2] + r * (c[3] + r * (c[4] + r * (
                    c[5] + r * (c[6] + r * (c[7] + r * c[8])))))))
            if y < 0:
                x = -x

%timeit -r5 -n3 BSM_Rational(N)
```

6min 4s ± 1.11 s per loop (mean ± std. dev. of 5 runs, 3 loops each)

[8]:
```
# Test the AR method with a exponential distribution
def AR_norm(size):
    count = 0
    while count < size:
        u1, u2 = np.random.uniform(size=2)
        y = -np.log(u2)
        if u1 <= np.exp(-(y - 1) ** 2 / 2):
            u3 = np.random.uniform()
            if u3 < 0.5:
                x = -y
            else:
                x = y
            count += 1

%timeit -r5 -n3 AR_norm(N)
```

```
18min 41s ± 3.22 s per loop (mean ± std. dev. of 5 runs, 3 loops each)
```

4 Problem 5

[9]:
```
# Define cards to be shuffled and the 'hit' order
cards = np.arange(1, 101)
order = np.arange(1, 101)
hits = []
# Simulate 10^8 times and calculate the mean and the variance
for i in range(int(N)):
    np.random.shuffle(cards)
    hits.append(np.sum(cards == order))

print('Expectation: {:.2f}'.format(np.mean(hits)))
print('Variance: {:.2f}'.format(np.var(hits)))
```

```
Expectation: 1.00
Variance: 1.00
```

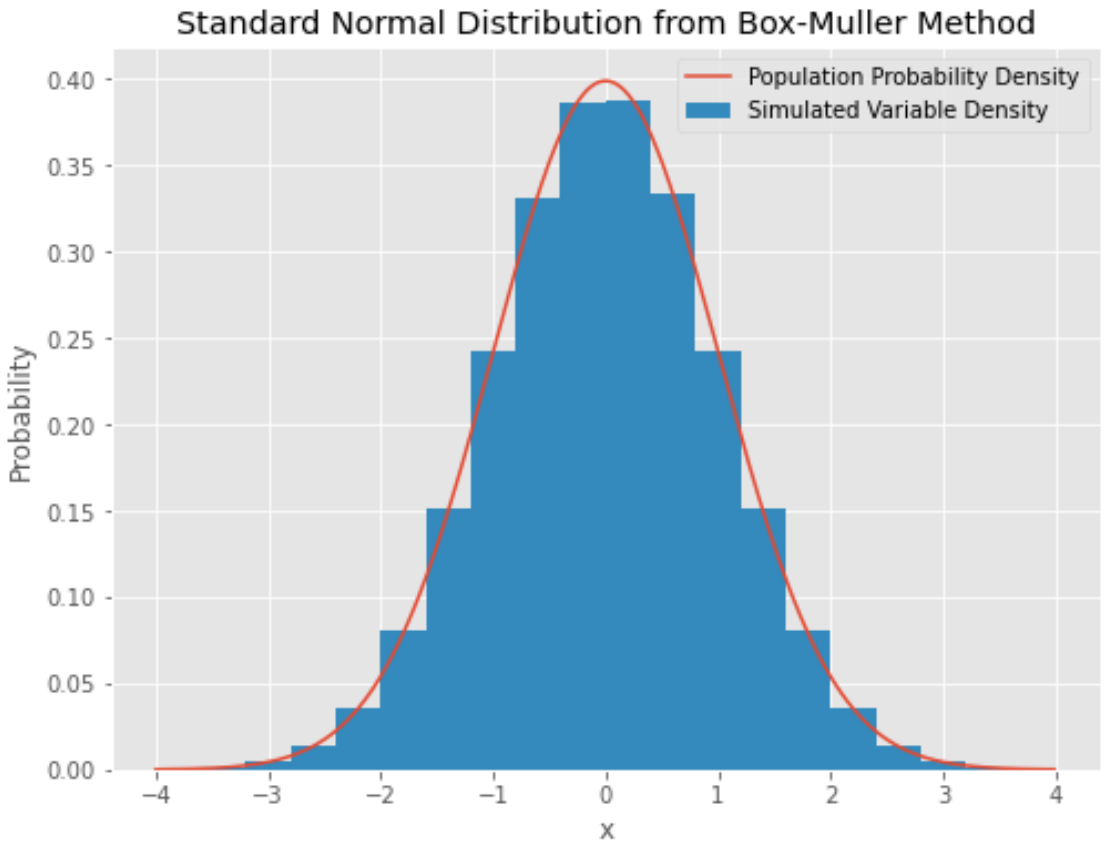
Appendix 2 - Algorithm Validation for Problem 4

January 31, 2022

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

# Specify the random seed and the plotting style
np.random.seed(124)
plt.style.use('ggplot')
```

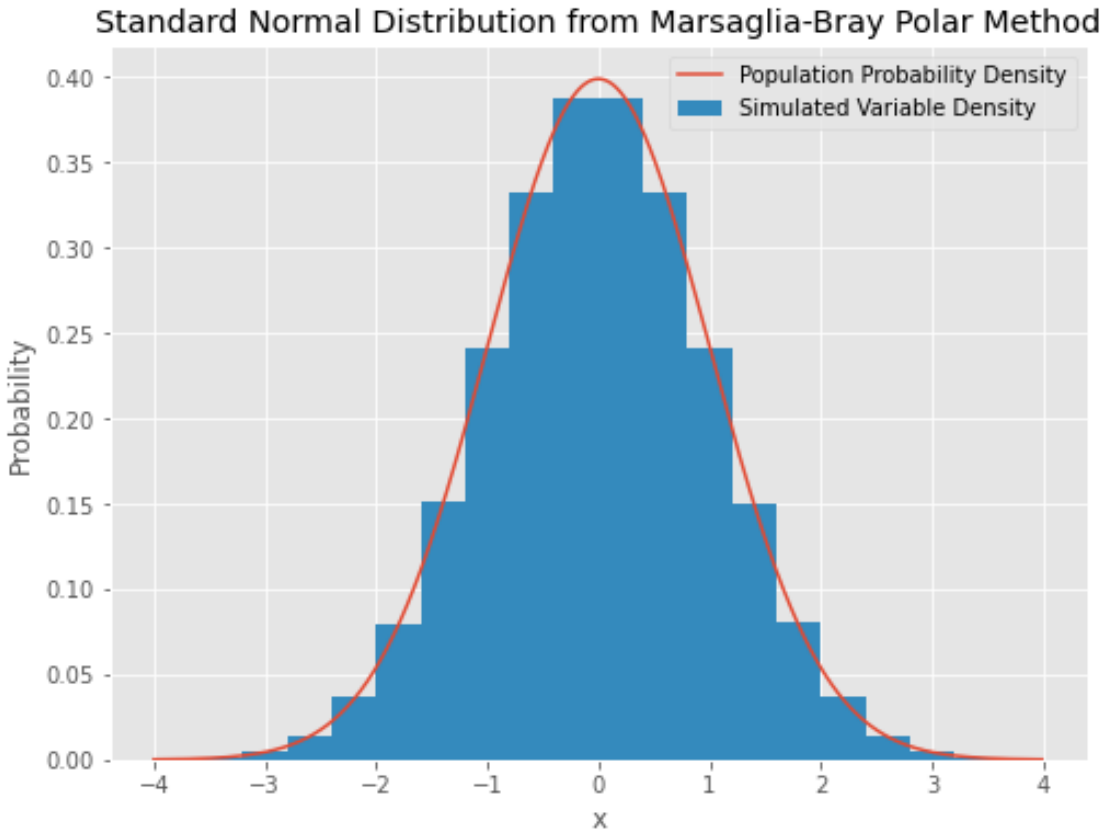
[2]:
```
# Define the sample size
N0 = int(1e6)

# Calculate the population density
x_true = np.arange(-4, 4, .01)
y_true = np.exp(-x_true ** 2 / 2) / np.sqrt(2 * np.pi)
```

[3]:
```
def BoxMuller(size, container):
    for i in range(size // 2):
        u1, u2 = np.random.uniform(size=2)
        param = np.sqrt(-2 * np.log(u1))
        x = param * np.cos(2 * np.pi * u2)
        y = param * np.sin(2 * np.pi * u2)
        # Store the generated r.v. into the container
        container.extend([x, y])

# Initialize the container
bm_norm = []
BoxMuller(N0, bm_norm)

# Plot
plt.figure(figsize=(8, 6))
plt.plot(x_true, y_true, label='Population Probability Density')
plt.hist(bm_norm, bins=20, density=True, range=(-4, 4),
         label='Simulated Variable Density')
plt.xlabel('x')
plt.ylabel('Probability')
plt.legend()
plt.title('Standard Normal Distribution from Box-Muller Method')
# plt.savefig('box_muller.png')
plt.show()
```



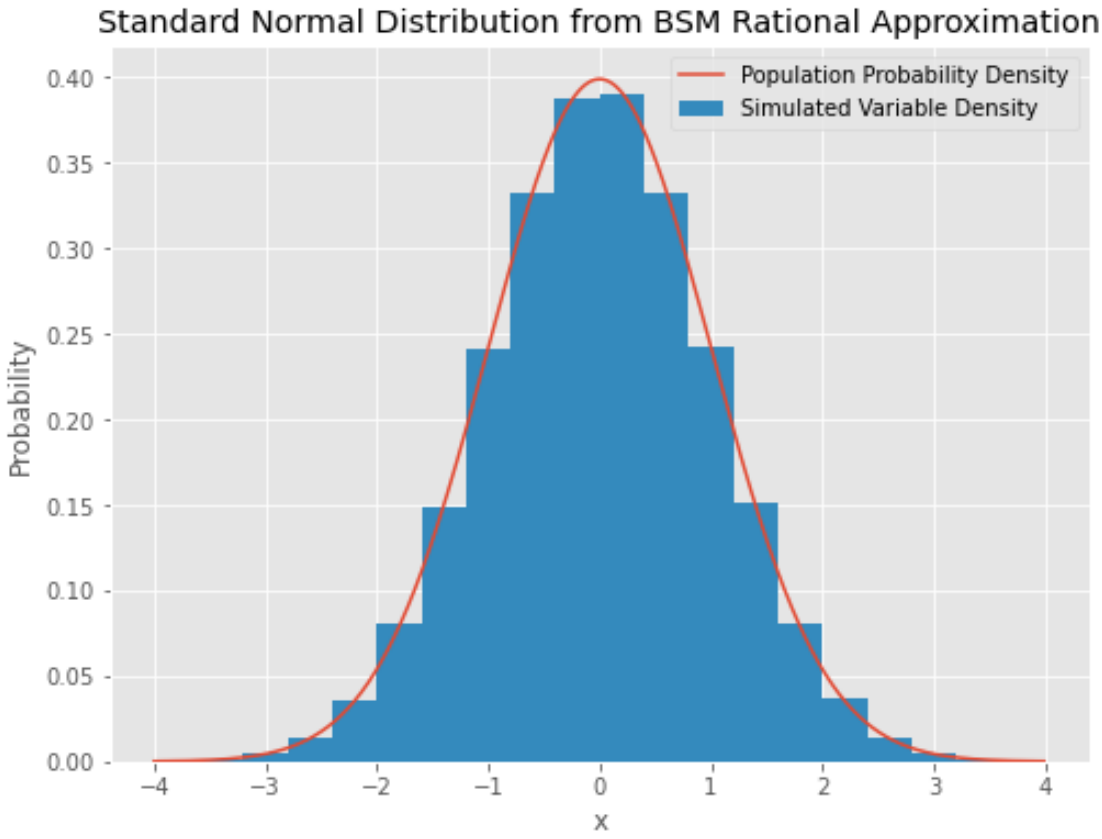
[4]:
```
def Marsaglia(size, container):
    count = 0
    while count < size:
        v1, v2 = np.random.uniform(-1, 1, 2)
        s = v1 ** 2 + v2 ** 2
        if s <= 1:
            param = np.sqrt(-2 * np.log(s) / s)
            x = v1 * param
            y = v2 * param
            container.extend([x, y])
            count += 2

mar_norm = []
Marsaglia(N0, mar_norm)

# Plot
plt.figure(figsize=(8, 6))
plt.plot(x_true, y_true, label='Population Probability Density')
plt.hist(mar_norm, bins=20, density=True, range=(-4, 4),
         label='Simulated Variable Density')
```

```
plt.xlabel('x')
plt.ylabel('Probability')
plt.legend()
plt.title('Standard Normal Distribution from Marsaglia-Bray Polar Method')
# plt.savefig('marsaglia.png')
plt.show()
```



[5]:
```
def BSM_Rational(size, container):
    a = [2.50662823884, -18.61500062529, 41.39119773534, -25.44106049637]
    b = [-8.47351093090, 23.08336743743, -21.06224101826, 3.13082909833]
    c = [0.3374754822726147, 0.9761690190917186, 0.1607979714918209,
         0.0276438810333863, 0.0038405729373609, 0.0003951896511919,
         0.0000321767881768, 0.0000002888167364, 0.0000003960315187]
    for i in range(size):
        u = np.random.uniform()
        y = u - 0.5
        if np.abs(y) < 0.42:
            r = y * y
            x = y * (((a[3] * r + a[2]) * r + a[1]) * r + a[0]) / (
                        (((b[3] * r + b[2]) * r + b[1]) * r + b[0]) * r + 1)
```

```
        else:
            r = u
            if y > 0:
                r = 1 - u
            r = np.log(-np.log(r))
            x = c[0] + r * (c[1] + r * (c[2] + r * (c[3] + r * (c[4] + r * (
                    c[5] + r * (c[6] + r * (c[7] + r * c[8])))))))
            if y < 0:
                x = -x
        container.append(x)

bsm_norm = []
BSM_Rational(N0, bsm_norm)

# Plot
plt.figure(figsize=(8, 6))
plt.plot(x_true, y_true, label='Population Probability Density')
plt.hist(bsm_norm, bins=20, density=True, range=(-4, 4),
         label='Simulated Variable Density')
plt.xlabel('x')
plt.ylabel('Probability')
plt.legend()
plt.title('Standard Normal Distribution from BSM Rational Approximation')
# plt.savefig('bsm_rational.png')
plt.show()
```



[6]:
```python
def AR_norm(size, container):
    count = 0
    while count < size:
        u1, u2 = np.random.uniform(size=2)
        y = -np.log(u2)
        if u1 <= np.exp(-(y - 1) ** 2 / 2):
            u3 = np.random.uniform()
            if u3 < 0.5:
                x = -y
            else:
                x = y
            container.append(x)
            count += 1

ar_norm = []
AR_norm(N0, ar_norm)

# Plot
plt.figure(figsize=(8, 6))
plt.plot(x_true, y_true, label='Population Probability Density')
```

```
plt.hist(ar_norm, bins=20, density=True, range=(-4, 4),
         label='Simulated Variable Density')
plt.xlabel('x')
plt.ylabel('Probability')
plt.legend()
plt.title('Standard Normal Distribution from AR Method')
# plt.savefig('ar.png')
plt.show()
```