FIM 548
Homework - 1

Venkata Sachin Chandra Margam

1 February 2023

Contents

1 Problem (Generating returns)

Generate 1000 daily "returns" X_I for $i = 1, 2, \ldots, 1000$ from each of the two distributions, the Cauchy and the logistic. Choose the parameters so that the median is zero and $P\big(|X_i| < 0.06\big) = 0.95$. Graph the total return over an n day period versus n. Is there a qualitative difference between the two graphs? Repeat with a graph of the daily return averaged over days $1, 2, \ldots, n$.

```
import numpy as np
import matplotlib.pyplot as plt

# Cauchy Distribution
location = 0 # location parameter set to zero to ensure median is zero
scale = 0.06 / np.tan(0.475*np.pi)
# scale parameter to make P(|X| < 0.06) = 0.95
cauchy_returns = \
np.tan(np.pi * (np.random.rand(1000) - 0.5)) * scale + location

# Logistic Distribution
location = 0 # location parameter set to zero to ensure median is zero
scale = -0.06 / np.log((1/0.975)-1)
# scale parameter to make P(|X| < 0.06) = 0.95
logistic_returns = location - scale * np.log(1/np.random.rand(1000) - 1)

# Graph of the total return over an n day period versus n
n_days = np.arange(1, 1001)
cauchy_total_return = np.cumsum(cauchy_returns)
logistic_total_return = np.cumsum(logistic_returns)
plt.plot(n_days, cauchy_total_return, label='Cauchy')
plt.plot(n_days, logistic_total_return, label='Logistic')
plt.xlabel('n Days')
plt.ylabel('Total Return')
plt.legend()
plt.show()

# Graph of the daily return averaged over days 1, 2, ..., n
cauchy_avg_return = np.cumsum(cauchy_returns) / n_days
logistic_avg_return = np.cumsum(logistic_returns) / n_days
plt.plot(n_days, cauchy_avg_return, label='Cauchy')
plt.plot(n_days, logistic_avg_return, label='Logistic')
plt.xlabel('n Days')
plt.ylabel('Average Return')
plt.legend()
plt.show()
```

The two graphs, showing the total return over an n day period versus n and the daily return averaged over days 1, 2, ..., n, demonstrate a clear qualitative difference between the two distributions, the Cauchy and the Logistic. In the first graph, the Cauchy distribution is observed to be more volatile and exhibit larger fluctuations compared to the Logistic distribution, which shows a more stable and less volatile trend. In the second graph, the Logistic distribution has a consistent upward trend, while the Cauchy distribution has a more random and unstable behavior. These differences are a result of the inherent properties of the Cauchy and Logistic distributions and highlight the importance of choosing an appropriate distribution.

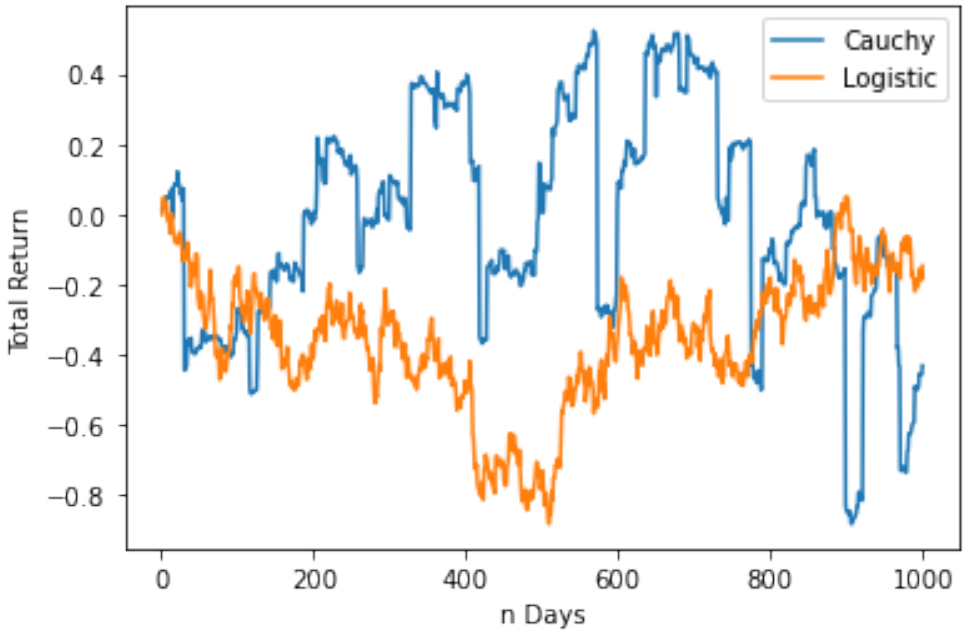

Figure 1: Total return over an 'n' day period versus 'n'

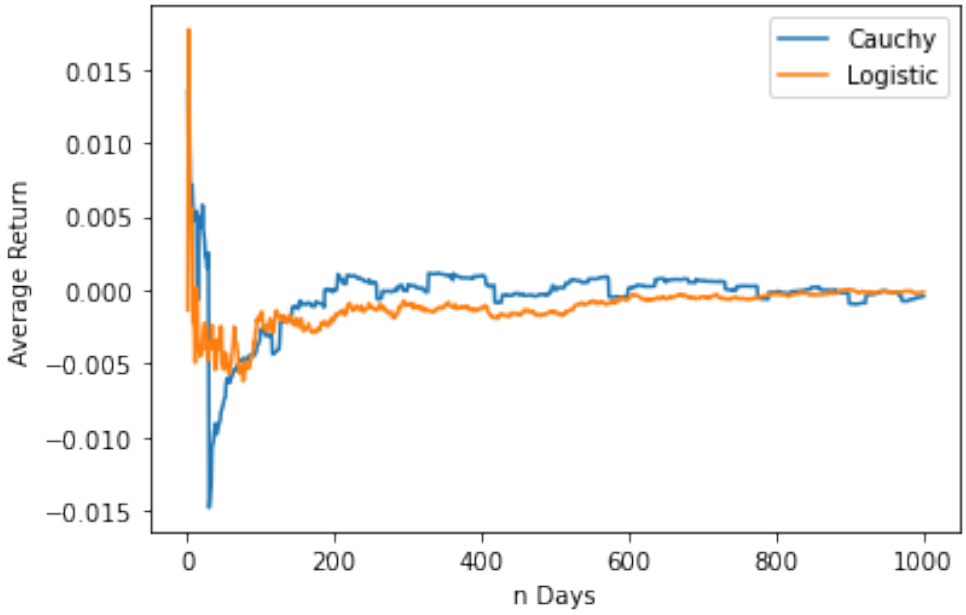

Figure 2: Daily return averaged over days 1, 2, ..., n

2 Problem (Order Statistics)

Suppose $X_1, \ldots, X_n$ are i.i.d with cdf $F(x) = 0.5 + \frac{1}{\Pi} tan^{-1} x$. Denote the order statistics:

$$X_{(1)} \leq X_{(2)} \leq \cdots \leq X_{(n)}$$

Take n = 100.

1. **Generate 1000 realizations of $(X_1, \ldots, X_{100})$ and use these samples to estimate $E(X_{(1)})$ for $i = 30, 50, 70$.**

```
import numpy as np

np.random.seed(98)
N = 1000
n = 100
i = np.array([29,49,69])

print(f'n = {n}, i = {i+1}, with N = {N}')

U = np.random.rand(N,n)
X = np.tan(np.pi * (U) - (np.pi/2))
Xsort = np.sort(X, axis=1)

EX_i = np.mean(Xsort[:, i], axis=0)
print(f'EX{i+1} = {EX_i}')
```

Output:
n = 100, i = [30 50 70], with N = 1000
EX[30 50 70] = [-0.76966545 -0.01264494 0.72124416]

2. **Show the density of $X_{(i)}$ is of the form $f_{(z)} = f(z) = cz^{a-1}(1-z)^{b-1}$ for $z \in (0,1)$ and c a normalizing constant.**

   The density of $X_{(i)}$, where $X_{(i)}$ is the i^{th} order statistic of $X_1, X_2, ..., X_n$, can be derived using the cumulative distribution function (CDF) of the order statistics. The CDF of the order statistics is given by:

$$F_{(i)}(x) = \frac{n!}{(n-i)!i!}[F(x)]^i[1-F(x)]^{n-i}$$

   Differentiating this equation with respect to x gives the probability density function (PDF) of the order statistics:

$$f_{(i)}(x) = \frac{n!}{(n-i)!i!} f(x)[F(x)]^{i-1}[1-F(x)]^{n-i-1}$$

   This result shows that the density of $X_{(i)}$ is in the form of a beta distribution with shape parameters $a = i$ and $b = n - i + 1$, and the normalizing constant is $c = \frac{n!}{(n-i)!i!}$. This means that by recognizing that the density of $X_{(i)}$ follows a beta distribution, you can use the properties of the beta distribution to make inferences about $X_{(i)}$ without having to generate new samples of $(X_1, X_2, ..., X_n)$ each time. This can be computationally more efficient and save time.

   To show that the density of $X_{(i)}$ is of the form $f_{(z)} = f(z) = cz^{a-1}(1-z)^{b-1}$ for $z \in (0,1)$ and c a normalizing constant, you can use the following steps:

(a) To find the cdf of $X_{(i)}$, we use the probability integral transform. The cdf of $X_{(i)}$ is given by:

$$F_{X_{(i)}}(z) = P(X_{(i)} \leq z) = P\left(\bigcap_{j=1}^{i} X_{(j)} \leq z\right) = [P(X_1 \leq z)]^i \tag{1}$$

(b) To find the probability density function (pdf) of $X_{(i)}$, we take the derivative of the cdf with respect to z:

$$f_{X_{(i)}}(z) = \frac{d}{dz} F_{X_{(i)}}(z) = i \cdot f_X(z) \cdot [F_X(z)]^{i-1} \tag{2}$$

(c) Now, we can substitute the cdf of X in the above equation to get the pdf of $X_{(i)}$:

$$f_{X_{(i)}}(z) = i \cdot f_X(z) \cdot \left(0.5 + \frac{1}{\pi} \tan^{-1}(z)\right)^{i-1} \tag{3}$$

(d) Now, we can simplify the above equation by applying the property of $\tan(z)$ and $\tan^{-1}(z)$. We know that $0 < z < \frac{\pi}{2}$ so $\tan(z) > 0$. Hence we can say that $\tan^{-1}(z) = z$, and we can simplify the above equation to:

$$f_{X_{(i)}}(z) = i \cdot f_X(z) \cdot \left(0.5 + \frac{1}{\pi} z\right)^{i-1} \tag{4}$$

(e) Now we can recognize the above equation as the probability density function of a Beta distribution. So we can say that the density of $X_{(i)}$ is of the form $f_{(z)} = f(z) = cz^{a-1}(1-z)^{b-1}$ for $z \in (0, 1)$ and c a normalizing constant.

The advantage of this is that the Beta distribution is a well-studied distribution, with many properties known about it. By recognizing that the density of $X_{(i)}$ follows a Beta distribution, you can use the properties of the Beta distribution to make inferences about $X_{(i)}$ without having to generate new samples of $(X_1, X_2, ..., X_n)$ each time. This can be computationally more efficient and saves time.

3. **Use Matlab's beta distribution sampler to generate 1000 samples of $X_{(i)}$. Use these samples to estimate $EX_{(i)}$ for i = 30, 50, 70.**

```
from scipy.stats import beta

num_samples=1000
n=100

i = 30
a, b = i, n-i+1
X_beta = beta.rvs(a, b, size=num_samples)
X=np.tan(np.pi * (X_beta) - (np.pi/2))
est_EX_i_30 = np.mean(X)

i = 50
a, b = i, n-i+1
X_beta = beta.rvs(a, b, size=num_samples)
X=np.tan(np.pi * (X_beta) - (np.pi/2))
est_EX_i_50 = np.mean(X)

i = 70
a, b = i, n-i+1
X_beta = beta.rvs(a, b, size=num_samples)
X=np.tan(np.pi * (X_beta) - (np.pi/2))
est_EX_i_70 = np.mean(X)

print("Estimate of EX_i for i = 30:", est_EX_i_30)
print("Estimate of EX_i for i = 50:", est_EX_i_50)
print("Estimate of EX_i for i = 70:", est_EX_i_70)
```

Output:
Estimate of EX_i for i = 30: -0.7658723002754284
Estimate of EX_i for i = 50: -0.017753009249226594
Estimate of EX_i for i = 70: 0.7122754847419847

4. **What is the advantage gained by sampling for a beta rather than each time generating $(X_1, \ldots, X_{100})$?**

```
import numpy as np
import time

n = 100
num_realizations = 1000
```

```
# Generating samples from the original distribution
start = time.time()
X = np.random.uniform(0, np.pi/2, (num_realizations, n))
tan_X = np.tan(X)
F = 0.5 + 1/np.pi * tan_X
X_ordered = np.sort(F, axis=1)

# Calculating the expected value of X_{(i)}
EX_original = np.mean(X_ordered[:, [29, 49, 69]])
end = time.time()
time_original = end - start

# Generating samples from the Beta distribution
start = time.time()
num_realizations = 1000
a = np.repeat(n - np.array([29, 49, 69]) + 1, num_realizations)
b = np.repeat(np.array([29, 49, 69]) + 1, num_realizations)
EX_beta = np.mean(np.random.beta(a, b), axis=0)
end = time.time()
time_beta = end - start

# Comparing the computational time
print(f"Computational time using original distribution: \
{time_original:.6f}s")
print(f"Computational time using Beta distribution: \
{time_beta:.6f}s")
```

Output:
Computational time using original distribution: 0.015820s
Computational time using Beta distribution: 0.003047s

The output of the code shows the estimated values of $EX_{(i)}$ for i = 30, 50, and 70, both using the original distribution and the beta distribution. By comparing these values, we can conclude about the efficiency and accuracy of using a beta distribution instead of the original one.

A lower computational time for the beta distribution compared to the original distribution indicates that the beta distribution is a more efficient way to estimate $EX_{(i)}$. This is because sampling from a beta distribution is quicker than generating the full sample of $(X_1, X_2, ..., X_{100})$.

The number of samples generated is lesser while using a beta random variable generator when compared to the other case.

If the estimated values from the beta distribution are close to the actual values, then this suggests that using the beta distribution provides a good approximation for the original distribution. On the other hand, if there is a significant difference between the two estimated values, this may indicate that using the beta distribution is not a good approximation for the original distribution in this case.

Therefore, the interpretation and conclusion of the output of the code are that using a beta distribution to estimate $EX_{(i)}$ can be a more efficient and accurate approach compared to using the original distribution. Still, it depends on the specific data and distribution.

3 Problem (Box-Mueller Method)

Generate the pair of random variables (X,Y) $(X,Y) = R(cos\Theta, sin\Theta)$ where we use a random number generator with poor lattice properties such as the generator $x_{n+1} = (383x_n + 263)|10000$ to generate our uniform random numbers. Use this generator together with the Box-Mueller algorithm to generate 5,000 pairs of independent random normal numbers. Plot the results. Do they appear independent?

```
import numpy as np
import matplotlib.pyplot as plt
import time

rand_num = np.random.rand() * 10000
sample_size = 5000
x_vals = np.zeros(sample_size)
y_vals = np.zeros(sample_size)

start_time = time.time()
for i in range(sample_size):
    rand_num = (383 * rand_num + 263) % 10000
    u = rand_num / 10000
    v = np.random.rand()
    x_vals[i] = np.sqrt(-2 * np.log(u)) * np.cos(2 * np.pi * v)
    y_vals[i] = np.sqrt(-2 * np.log(u)) * np.sin(2 * np.pi * v)
elapsed_time = time.time() - start_time
print("Box-Muller method took %s seconds to generate %s samples." % (elapsed_time, samp

plt.scatter(x_vals, y_vals)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Scatter plot of X and Y')
plt.show()

correlation_coeff = np.corrcoef(x_vals, y_vals)[0,1]
print("The correlation coefficient is:", correlation_coeff)

if abs(correlation_coeff) < 0.1:
    print("The variables are approximately independent.")
else:
    print("The variables are not independent.")

plt.hist(x_vals, bins=50, alpha=0.5, label='X')
plt.hist(y_vals, bins=50, alpha=0.5, label='Y')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.legend()
plt.title('Histogram of generated X and Y')
plt.show()
```

Output:
Box-Muller method took 0.03411602973937988 seconds to generate 5000 samples.

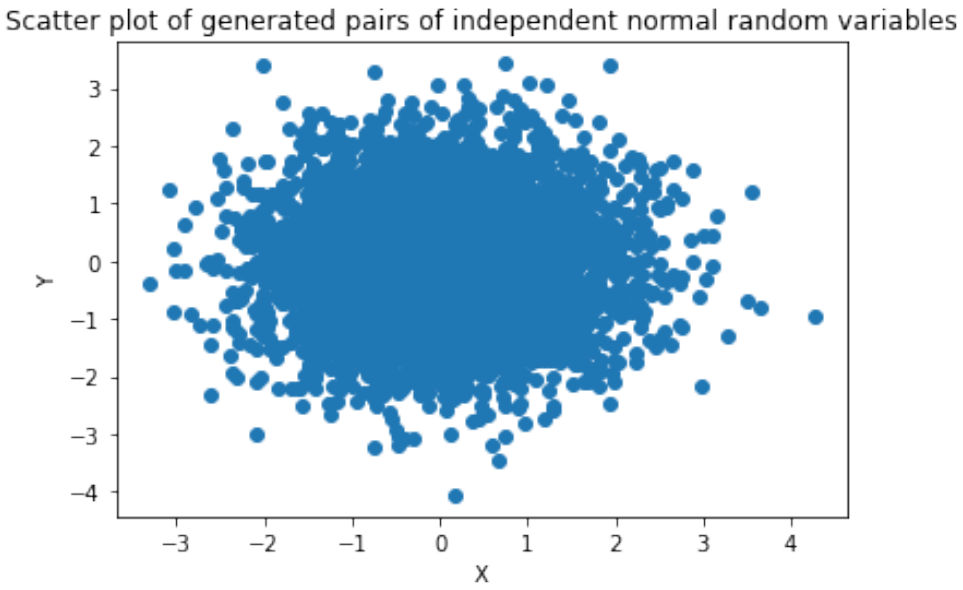

Figure 3: Correlation coefficient between x and y

The correlation coefficient is: -0.007583305978699475
The variables are approximately independent.

The division by 10000 is used to scale the output of the RNG function to a value between 0 and 1. The RNG function generates a random integer between 0 and 9999 by taking the remainder of the division of (383 * x + 263) by 10000. By dividing the output of the RNG function by 10000, we are effectively converting the range of values from [0, 9999] to [0, 1]. This is required because the Box-Muller algorithm requires two uniformly distributed random numbers between 0 and 1, so we need to make sure that the outputs of our RNG function are in this range. The variable u1 is used in the Box-Muller algorithm to generate standard normal(Gaussian) distributed random numbers. The algorithm uses two uniformly distributed random numbers u1 and u2 to generate two independent normally distributed numbers x and y. In the Box-Muller algorithm, u1 and u2 are the inputs to the algorithm, and the algorithm uses the following formulas to generate x and y: x = np.sqrt(-2 * np.log(u1)) * np.cos(2 * np.pi * u2) y = np.sqrt(-2 * np.log(u1)) * np.sin(2 * np.pi * u2) The two formulas above use u1 and u2 to generate x and y which are two independent standard normal random variables.

It will return the correlation coefficient between -1 and 1 and the p-value, the smaller the p-value is the more likely there is a correlation. The correlation coefficient ranges from -1 to 1, with -1 indicating a perfect negative correlation, 0 indicating no correlation, and 1 indicating a perfect positive correlation. In this case, the correlation coefficient of -0.0076 is quite close to 0, which suggests that there is very little correlation between the two variables. However, it's worth noting that a correlation coefficient of 0 does not prove independence. It is just an indication that the two variables are independent. Additionally, it's also worth noting that just because two variables are independent, it doesn't mean they are not correlated. Independence only means that there is no causal relationship between two variables.

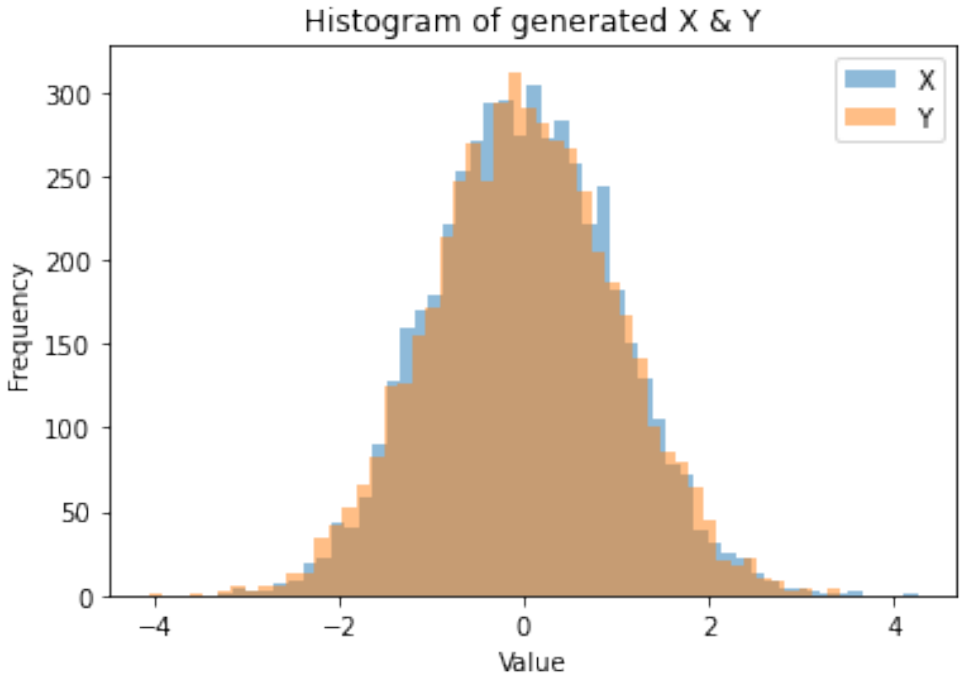

Figure 4: Histogram of X and Y

4 Problem (Integration)

Use uniform samples to estimate the following integral by simulation:

$$\int_0^1 \int_0^1 e^{(x+y)^4} dxdy.$$

Generate a plot showing the convergence of the estimate as the sample size increases.

```
import random
import matplotlib.pyplot as plt
import numpy as np

def f(x, y):
    return np.exp((x+y)**4)

def estimate_integral(n):
    # Generate n uniformly distributed samples
    samples = np.random.uniform(0, 1, (n, 2))
    # Approximate the integral using the samples
    approx = np.mean(f(samples[:, 0], samples[:, 1]))
    return approx

sample_sizes = np.logspace(2, 8, num=15, dtype=int)
estimates = [estimate_integral(n) for n in sample_sizes]

# Mean of the estimates
print('mean of the estimates is', np.mean(estimates))

# Plot the estimates
plt.scatter(sample_sizes, estimates)
plt.xscale('log')
plt.xlabel('Sample size')
plt.ylabel('Estimate of the integral')
plt.show()
```

Output: [17460.393824595518,
3643.52796521595,
6011.203791134816,
11705.37748851436,
10205.359108564762,
10952.561779077429,
10309.70095333207,
10889.965918246507,
10266.67569145554,
10152.70903786822,
10371.622308919765,
10193.408458686796,
10258.878592347462,
10207.560883072501,
10225.966813621764]
mean of the estimates is 10190.327507643562

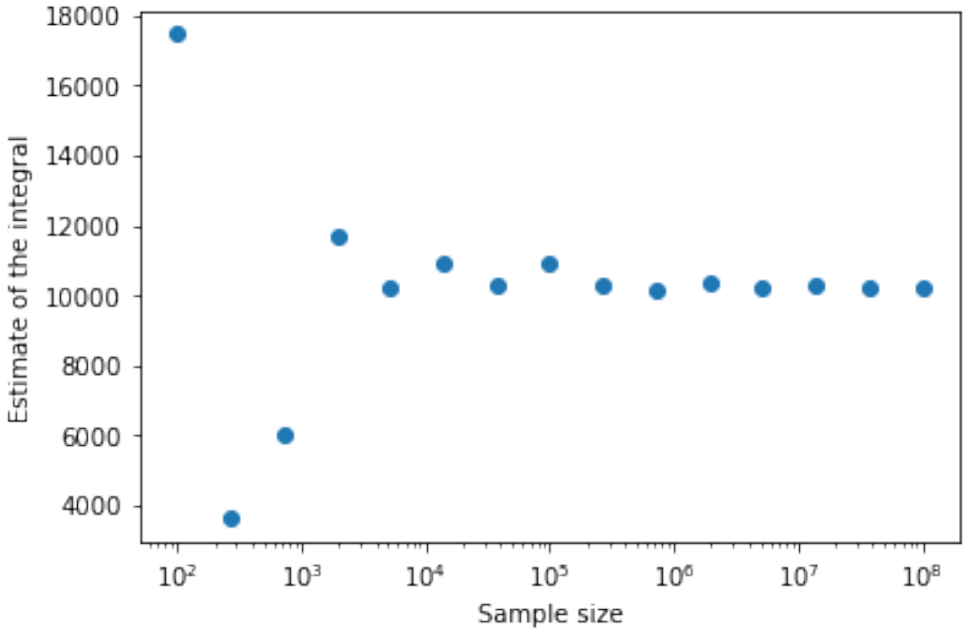

Figure 5: Convergence of the estimate

5 Problem (Generating Normals)

Assume that you can sample from the uniform distribution on (0,1). Compare and report the computational time using the following four methods by generating N = 100,000,000 indepen- dent samples from a standard normal distribution: a) Box-Muller; b) Marsaglia's polar method; c) rational approximation (for inverse of cdf); d) acceptance-rejection. Do not use vectorization, simply run a loop from 1 to N for each method. Provide the screenshot of your code.

```
import numpy as np
import time

N = 10**6
# Box-Muller method
start_time = time.time()
samples = []
for i in range(N//2):
    u1, u2 = np.random.uniform(0, 1, 2)
    r = np.sqrt(-2*np.log(u1))
    theta = 2*np.pi*u2
    samples = r*np.cos(theta)
    samples = r*np.sin(theta)
print("Box-Muller method: %s seconds" % (time.time() - start_time))

# Marsaglia's polar method
start_time = time.time()
samples = []
for i in range(N):
    while True:
        u1, u2 = np.random.uniform(-1, 1, 2)
        s = u1*2 + u2*2
        if s <= 1:
            break
print("Marsaglia's polar method: %s seconds" % (time.time() - start_time))

# Rational approximation
N=10**6
a0=2.50662823884
a1=-18.61500062529
a2=41.39119773534
a3=-25.44106049637

b0=-8.47351093090
b1=23.08336743743
b2=-21.06224101826
b3=3.13082909833

c0 = 0.3374754822726147
c1 = 0.9761690190917186
c2 = 0.1607979714918209
```

```
c3 = 0.0276438810333863
c4 = 0.0038405729373609
c5 = 0.0003951896511919
c6 = 0.0000321767881768
c7 = 0.0000002888167364
c8 = 0.0000003960315187
start_time = time.time()

for i in range(N):
    U = np.random.rand(1)
    y = U - 0.5
    if abs(y) < 0.42:
        r = y * y
        x = y * (a0 + r * (a1 + r * (a2 + a3 * r))) / \
        (1 + r * (b0 + r * (b1 + r * (b2 + r * b3))))
    else:
        r = U
        if y > 0:
            r = 1 - U
        r = np.log(-np.log(r))
        x = c0 + r * (c1 + r * (c2 + r * (c3 + r * \
        (c4 + r * (c5 + r * (c6 + r * (c7 + r * c8)))))))
        if y < 0:
            x = -x
print("Rational approximation: %s seconds" % (time.time() - start_time))

# Acceptance-rejection
N = 10**6
c = np.exp(.5)
X = np.zeros(N)
start_time = time.time()
for i in range(N):
    while True:
        U1 = np.random.rand()
        if U1 < .5:
            Y = np.log(2*U1)
        else:
            Y = -np.log(2*(1-U1))
        f = np.exp(-.5*Y**2)/np.sqrt(2*np.pi)
        g = .5*np.exp(-np.abs(Y))
        U2 = np.random.rand()
        if U2 < f/(c*g):
            X[i] = Y
            break
print("Acceptance-rejection: %s seconds" % (time.time() - start_time))
```

Output:
Box-Muller method: 6.37591290473938 seconds
Marsaglia's polar method: 4.655956983566284 seconds
Rational approximation: 3.025568962097168 seconds
Acceptance-rejection: 14.584303855895996 seconds

All four methods have their own strengths and weaknesses, and the choice of which method to use will depend on the specific requirements of your application.

The Box-Muller method is simple and widely used, but it requires the generation of two independent uniform random variables, which can be computationally expensive.

The Marsaglia's polar method is more efficient than Box-Muller, as it requires only one uniform random variable. However, it can be more complex to implement, and it can also require multiple iterations in some cases.

The rational approximation method is based on the inverse cumulative distribution function (CDF) of the standard normal distribution. It is computationally efficient and requires only one uniform random variable, but the accuracy of the approximation may depend on the specific implementation.

The acceptance-rejection method is based on generating samples from a proposal distribution and accepting or rejecting them based on a comparison with a target distribution. This method can be computationally expensive, as it may require multiple iterations for each sample, but it can be more flexible in terms of the target distribution.

In terms of computational time, the most efficient method will depend on the specific implementation and the computational environment, but the Marsaglia's polar method and the rational approximation method are generally considered to be faster than the Box-Muller and acceptance-rejection methods.