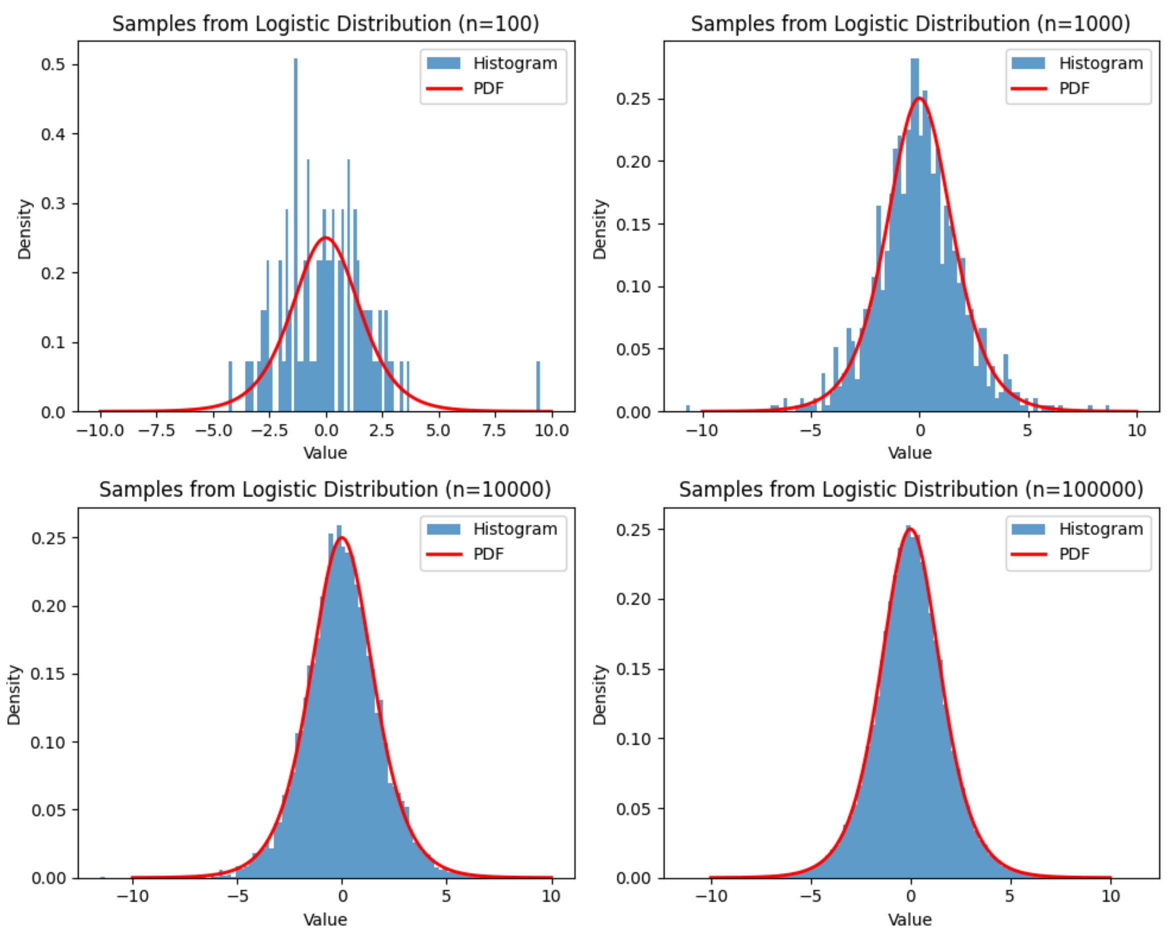
Problem1

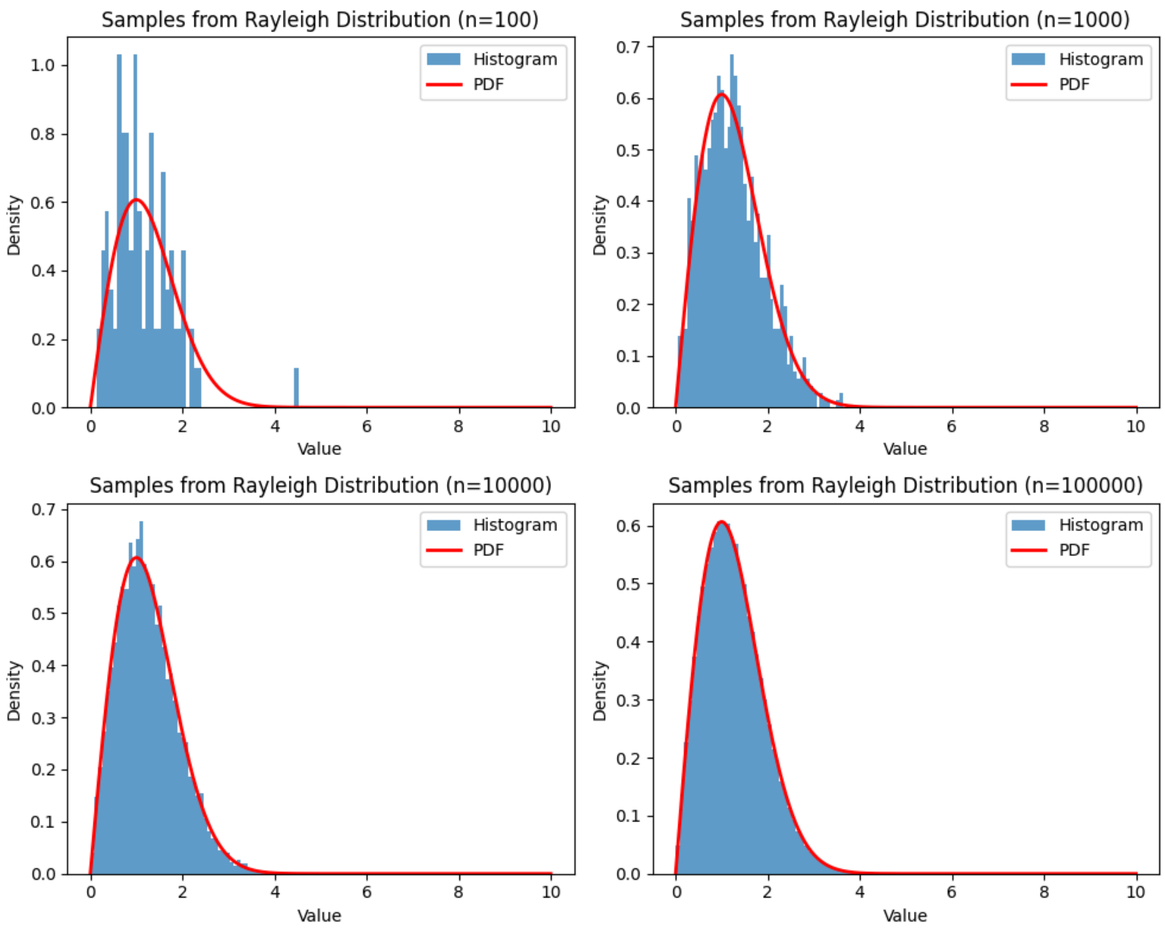
(a) The CDF of logistic distribution can be represented by $F(x) = \frac{e^x}{1+e^x}$, ∀x ∈ R, which is continuous and strictly increasing on the support of the distribution. So we can obtain its inverse function $F^{-1}(x) = -log(1/x - 1)$ and the PDF is $f(x) = \frac{e^x}{(1+e^x)^2}$.

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

# Define the inverse CDF of the logistic distribution
def inverse_cdf(u):
    return -np.log(1/u - 1)

# Define the PDF of the logistic distribution
def pdf(x):
    return np.exp(x) / ((1 + np.exp(x)) ** 2)

# Set the number of samples
n_values = [100, 1000, 10000, 100000]

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

for i, n in enumerate(n_values):
    # Generate random numbers between 0 and 1
    u = np.random.random(n)

    # Compute the samples using inverse transform sampling
    samples = inverse_cdf(u)

    # Calculate the subplot indices
    row = i // 2
    col = i % 2

    # Plot the histogram of the samples
    axes[row, col].hist(samples, bins=100, density=True, alpha=0.7, label='Histo

    # Plot the PDF of the logistic distribution
    x = np.linspace(-10, 10, 1000)
    axes[row, col].plot(x, pdf(x), 'r', linewidth=2, label='PDF')

    axes[row, col].set_xlabel('Value')
    axes[row, col].set_ylabel('Density')
    axes[row, col].set_title(f'Samples from Logistic Distribution (n={n})')
    axes[row, col].legend()

# Adjust the spacing between subplots
plt.tight_layout()

# Show the figure
plt.show()
```



(b) The CDF $F(x) = 1 - e^{-x^2/2}$, ∀x > 0, which is continuous and strictly increasing on the support of the distribution. So we can obtain its inverse function $F^{-1}(x) = \sqrt{-2log(1-y)}$ and the PDF is $f(x) = xe^{-x^2/2}$..

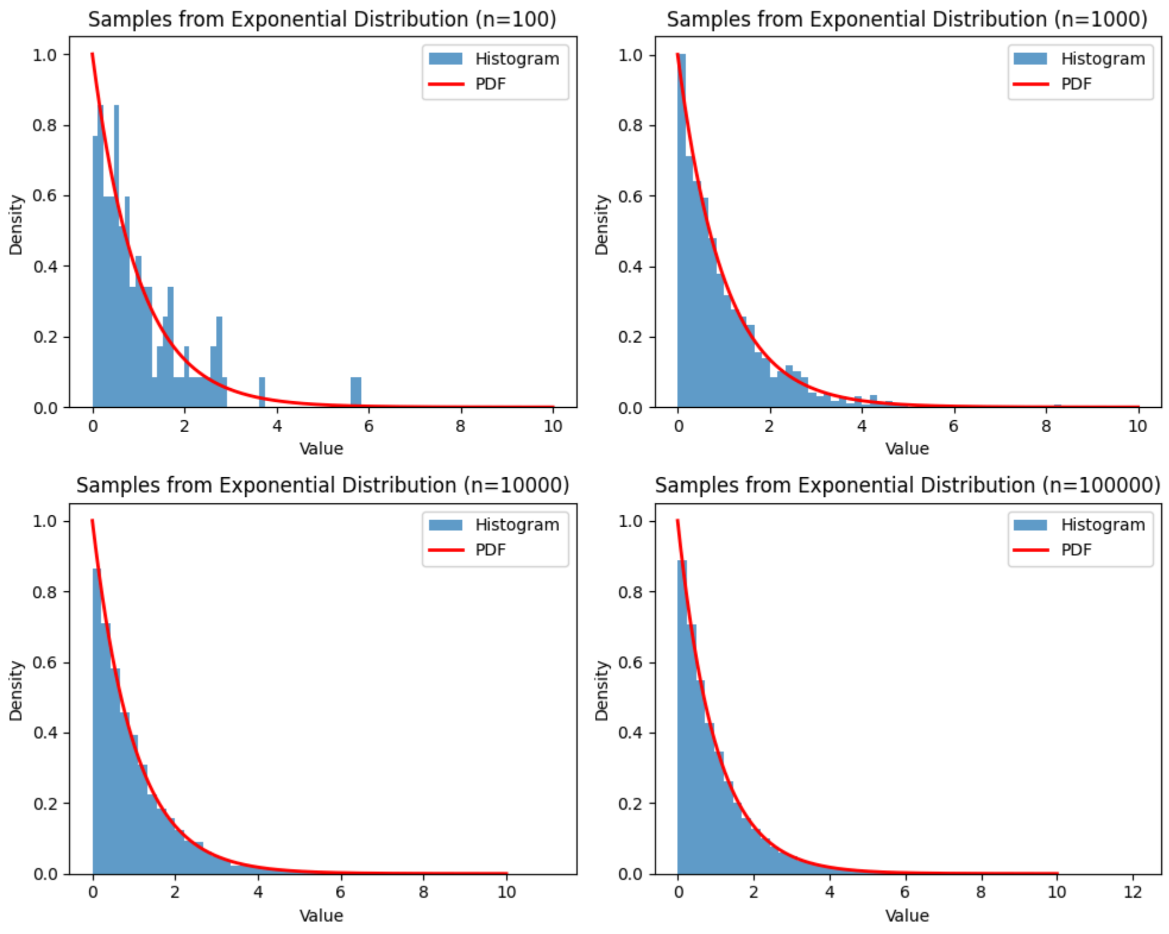
In [ ]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Define the inverse CDF of the logistic distribution
def inverse_cdf(u):
    return np.sqrt(-2 * np.log(1 - u))

# Define the PDF of the logistic distribution
def pdf(x):
    return x * np.exp(-x**2 / 2)

# Set the number of samples
n_values = [100, 1000, 10000, 100000]

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

for i, n in enumerate(n_values):
    # Generate random numbers between 0 and 1
    u = np.random.random(n)

    # Compute the samples using inverse transform sampling
    samples = inverse_cdf(u)

    # Calculate the subplot indices
    row = i // 2
    col = i % 2

    # Plot the histogram of the samples
    axes[row, col].hist(samples, bins=50, density=True, alpha=0.7, label='Histog

    # Plot the PDF of the logistic distribution
    x = np.linspace(0, 10, 1000)
    axes[row, col].plot(x, pdf(x), 'r', linewidth=2, label='PDF')

    axes[row, col].set_xlabel('Value')
    axes[row, col].set_ylabel('Density')
    axes[row, col].set_title(f'Samples from Rayleigh Distribution (n={n})')
    axes[row, col].legend()

# Adjust the spacing between subplots
plt.tight_layout()

# Show the figure
plt.show()
```



(c) The CDF $F(x) = 1 - e^{-x}$, ∀x > 0, which is continuous and strictly decreasing on the support of the distribution. So we can obtain its inverse function $F^{-1}(x) = -log(1-y)$ and the PDF is $f(x) = e^{-x}$.

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

# Define the inverse CDF of the logistic distribution
def inverse_cdf(u):
    return -np.log(1 - u)
```

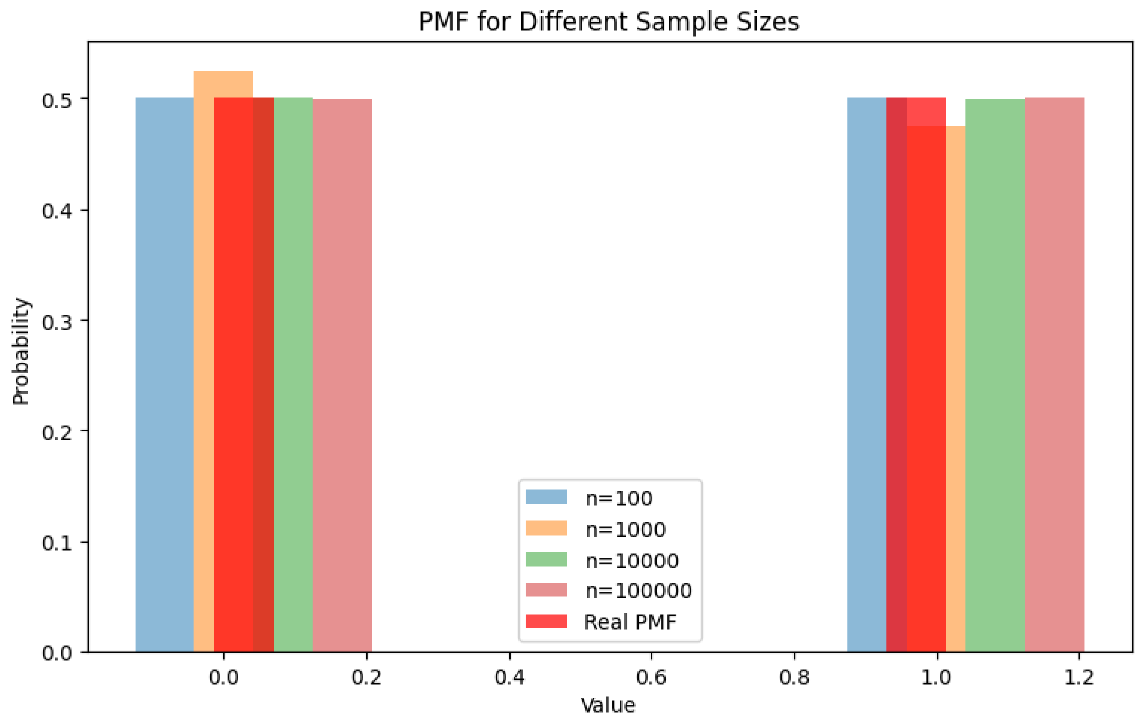
```python
# Define the PDF of the logistic distribution
def pdf(x):
    return np.exp(-x)

# Set the number of samples
n_values = [100, 1000, 10000, 100000]

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

for i, n in enumerate(n_values):
    # Generate random numbers between 0 and 1
    u = np.random.random(n)

    # Compute the samples using inverse transform sampling
    samples = inverse_cdf(u)

    # Calculate the subplot indices
    row = i // 2
    col = i % 2

    # Plot the histogram of the samples
    axes[row, col].hist(samples, bins=50, density=True, alpha=0.7, label='Histog

    # Plot the PDF of the logistic distribution
    x = np.linspace(0, 10, 1000)
    axes[row, col].plot(x, pdf(x), 'r', linewidth=2, label='PDF')

    axes[row, col].set_xlabel('Value')
    axes[row, col].set_ylabel('Density')
    axes[row, col].set_title(f'Samples from Exponential Distribution (n={n})')
    axes[row, col].legend()

# Adjust the spacing between subplots
plt.tight_layout()

# Show the figure
plt.show()
```

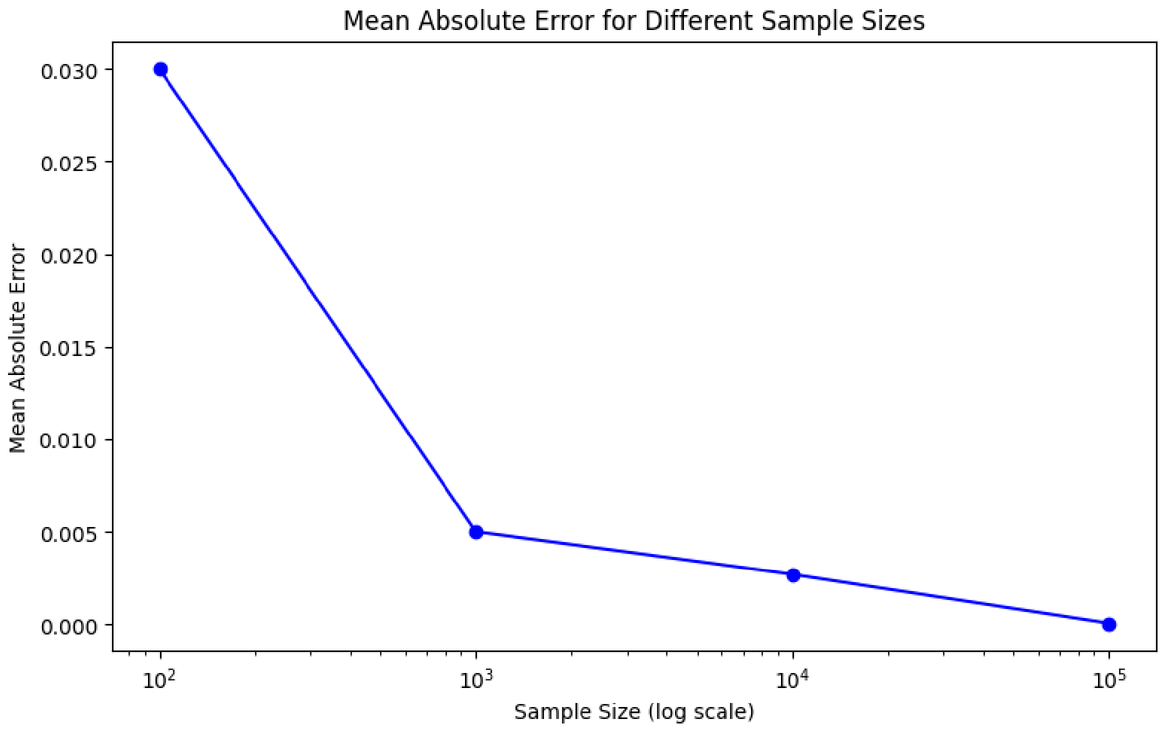


Problem 2

In [ ]:

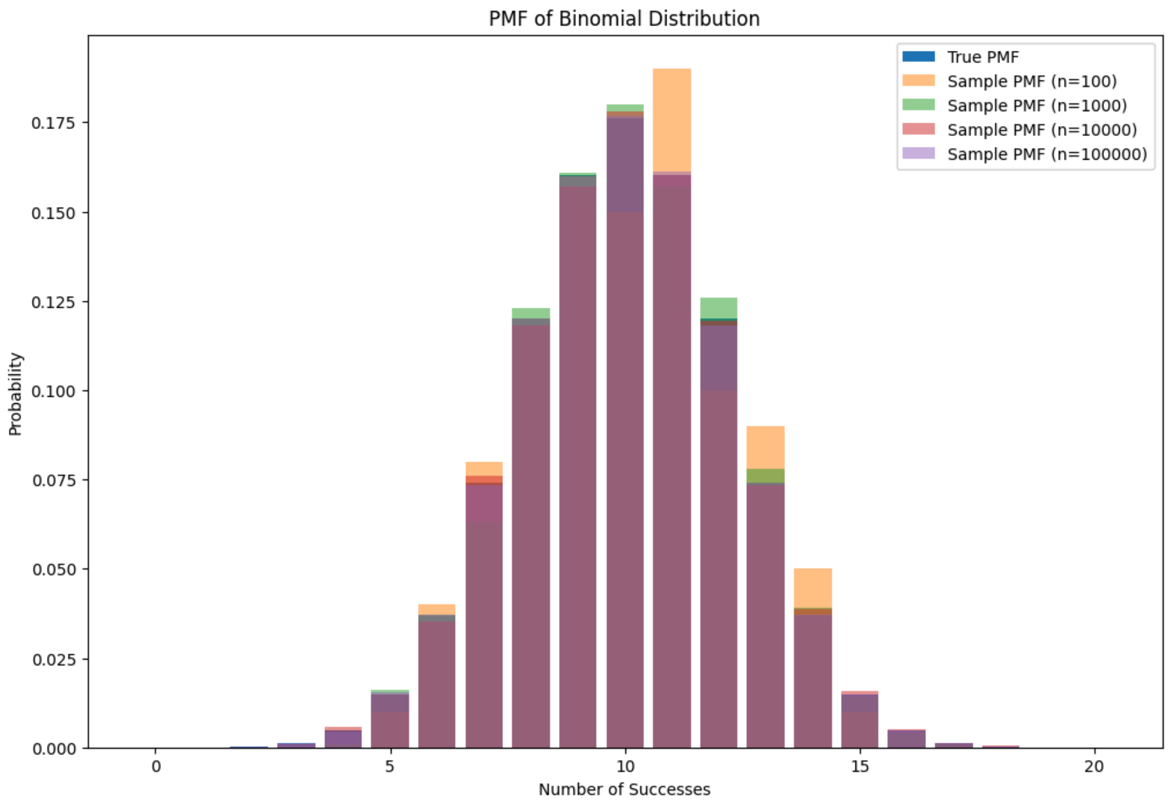
```python
import numpy as np
import matplotlib.pyplot as plt

def inverse_transform_sampling_bernoulli(p, n):
    samples = np.zeros(n)
    for i in range(n):
        u = np.random.random()
        samples[i] = 0 if u <= p else 1
    return samples

p = 0.5  # Probability of success
sample_sizes = [1e2, 1e3, 1e4, 1e5]  # Different sample sizes

# Generate real PMF
x = [0, 1]
real_pmf = [1-p, p]

# Plot PMF for different sample sizes
fig, axs = plt.subplots(2, 1, figsize=(8, 10))

bar_width = 0.25 / 3
bar_offset = -0.25 / 3

for i, size in enumerate(sample_sizes):
    size = int(size)
    samples = inverse_transform_sampling_bernoulli(p, size)
    unique, counts = np.unique(samples, return_counts=True)
    pmf = counts / size
    axs[0].bar(unique+i*bar_width+bar_offset, pmf, width=bar_width, alpha=0.5, l

# Plot real PMF
axs[0].bar([x[0]+bar_width/3, x[1]-bar_width/3], [real_pmf[0], real_pmf[1]], wic

axs[0].set_xlabel('Value')
axs[0].set_ylabel('Probability')
axs[0].set_title('PMF for Different Sample Sizes')
axs[0].legend()

# Plot error
errors = []
for size in sample_sizes:
    size = int(size)
    samples = inverse_transform_sampling_bernoulli(p, size)
    unique, counts = np.unique(samples, return_counts=True)
    pmf = counts / size
    error = np.abs(pmf - real_pmf).mean()
    errors.append(error)

axs[1].plot(sample_sizes, errors, 'bo-')
axs[1].set_xscale('log')
axs[1].set_xlabel('Sample Size (log scale)')
axs[1].set_ylabel('Mean Absolute Error')
axs[1].set_title('Mean Absolute Error for Different Sample Sizes')

plt.tight_layout()
plt.show()
```

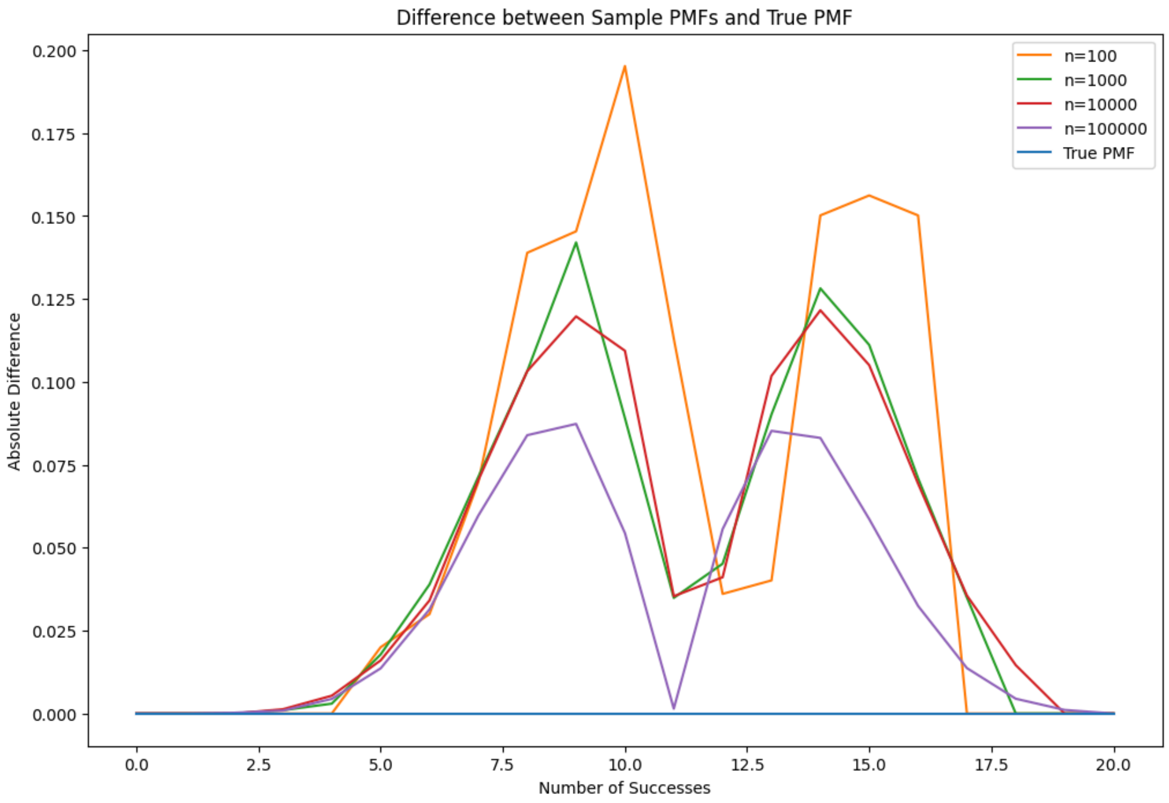




In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

def binomial_pmf(n, p, k):
    """Calculate the probability mass function (PMF) of the binomial distributio
    return np.math.comb(n, k) * (p ** k) * ((1 - p) ** (n - k))

def binomial_inverse_transform(n, p, size):
    """Generate samples from a binomial distribution using inverse transform sam
    u = np.random.uniform(size=size)  # Generate uniform random variables
    cdf = np.cumsum([binomial_pmf(n, p, k) for k in range(n + 1)])  # Calculate
    samples = np.searchsorted(cdf, u)  # Apply quantile function
    return samples

n = 20  # Number of trials
```

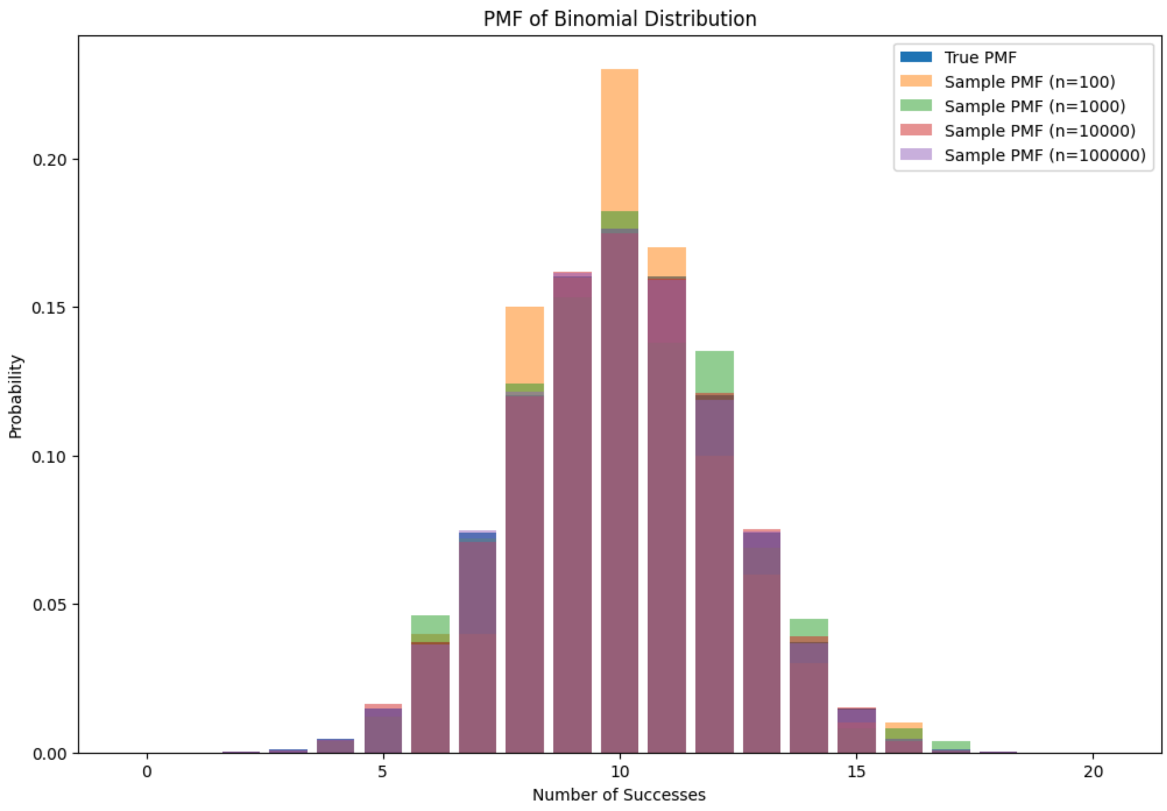
```python
p = 0.5  # Probability of success
sample_sizes = [100, 1000, 10000, 100000]

plt.figure(figsize=(12, 8))

# Set custom colors with enhanced contrast
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd']

# Plot the true PMF as a bar plot
x = np.arange(n + 1)
pmf = [binomial_pmf(n, p, k) for k in range(n + 1)]
plt.bar(x, pmf, color=colors[0], label='True PMF')

# Generate and plot samples for different sizes
for i, size in enumerate(sample_sizes):
    samples = binomial_inverse_transform(n, p, size)
    unique, counts = np.unique(samples, return_counts=True)
    sample_pmf = counts / size
    plt.bar(unique, sample_pmf, alpha=0.5, color=colors[i+1], label=f'Sample PMF

plt.xlabel('Number of Successes')
plt.ylabel('Probability')
plt.title('PMF of Binomial Distribution')
plt.legend()

# Calculate the differences between sample PMFs and the true PMF

# Plot the differences
plt.figure(figsize=(12, 8))

# Calculate the differences
sample_sizes = [100, 1000, 10000, 100000]
differences = []
for size in sample_sizes:
    samples = binomial_inverse_transform(n, p, size)
    unique, counts = np.unique(samples, return_counts=True)
    sample_pmf = counts / size
    difference = np.abs(sample_pmf - pmf[:len(unique)])
    all_values = np.arange(n + 1)
    difference_extended = np.zeros(n + 1)
    difference_extended[unique] = difference
    differences.append(difference_extended)

# Plot the differences
for i, size in enumerate(sample_sizes):
    plt.plot(all_values, differences[i], linestyle='-', color=colors[i+1], label

# Plot the true PMF
plt.plot(x, np.zeros_like(x), linestyle='-', color=colors[0], label='True PMF')

plt.xlabel('Number of Successes')
plt.ylabel('Absolute Difference')
plt.title('Difference between Sample PMFs and True PMF')
plt.legend()
plt.show()
```

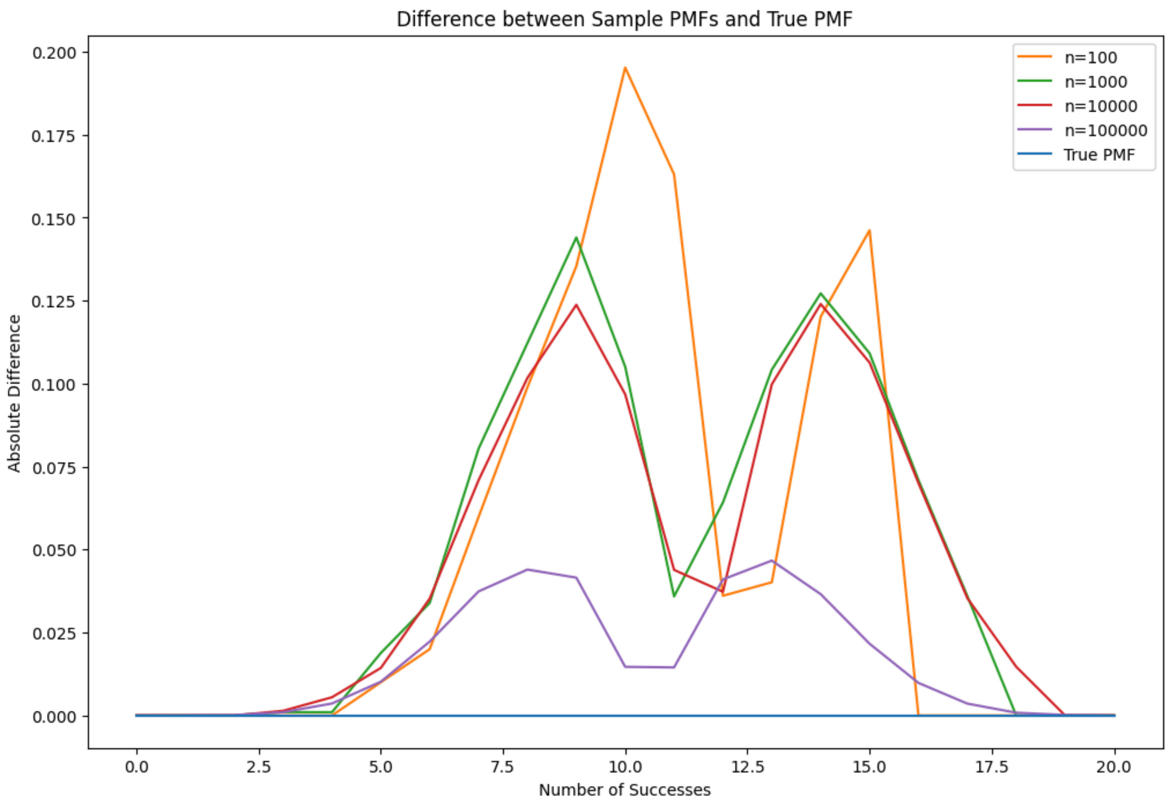




```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt

        def binomial_pmf(n, p, k):
            """Calculate the probability mass function (PMF) of the binomial distributio
            return np.math.comb(n, k) * (p ** k) * ((1 - p) ** (n - k))

        def binomial_inverse_transform(n, p, size):
            """Generate samples from a binomial distribution using inverse transform sam
            u = np.random.uniform(size=size)  # Generate uniform random variables
            cdf = np.cumsum([binomial_pmf(n, p, k) for k in range(n + 1)])  # Calculate
```

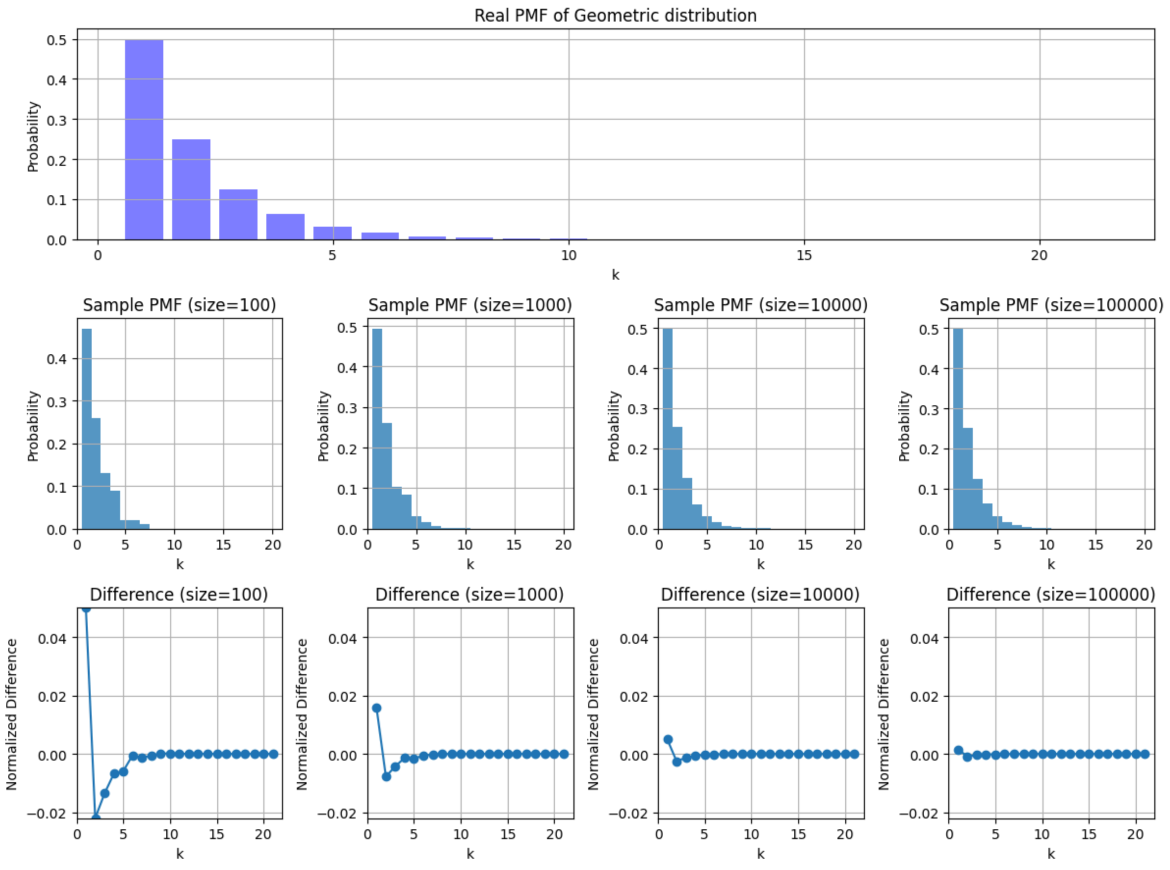
```
    samples = np.searchsorted(cdf, u)  # Apply quantile function
    return samples

n = 20  # Number of trials
p = 0.5  # Probability of success
sample_sizes = [100, 1000, 10000, 100000]

plt.figure(figsize=(12, 8))

# Set custom colors with enhanced contrast
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd']

# Plot the true PMF as a bar plot
x = np.arange(n + 1)
pmf = [binomial_pmf(n, p, k) for k in range(n + 1)]
plt.bar(x, pmf, color=colors[0], label='True PMF')

# Generate and plot samples for different sizes
for i, size in enumerate(sample_sizes):
    samples = binomial_inverse_transform(n, p, size)
    unique, counts = np.unique(samples, return_counts=True)
    sample_pmf = counts / size
    plt.bar(unique, sample_pmf, alpha=0.5, color=colors[i+1], label=f'Sample PMF

plt.xlabel('Number of Successes')
plt.ylabel('Probability')
plt.title('PMF of Binomial Distribution')
plt.legend()

# Calculate the differences between sample PMFs and the true PMF

# Plot the differences
plt.figure(figsize=(12, 8))

# Calculate the differences
sample_sizes = [100, 1000, 10000, 100000]
differences = []
for size in sample_sizes:
    samples = binomial_inverse_transform(n, p, size)
    unique, counts = np.unique(samples, return_counts=True)
    sample_pmf = counts / size
    difference = np.abs(sample_pmf - pmf[:len(unique)])
    all_values = np.arange(n + 1)
    difference_extended = np.zeros(n + 1)
    difference_extended[unique] = difference
    differences.append(difference_extended)

# Plot the differences
for i, size in enumerate(sample_sizes):
    plt.plot(all_values, differences[i], linestyle='-', color=colors[i+1], label

# Plot the true PMF
plt.plot(x, np.zeros_like(x), linestyle='-', color=colors[0], label='True PMF')

plt.xlabel('Number of Successes')
plt.ylabel('Absolute Difference')
plt.title('Difference between Sample PMFs and True PMF')
plt.legend()
plt.show()
```





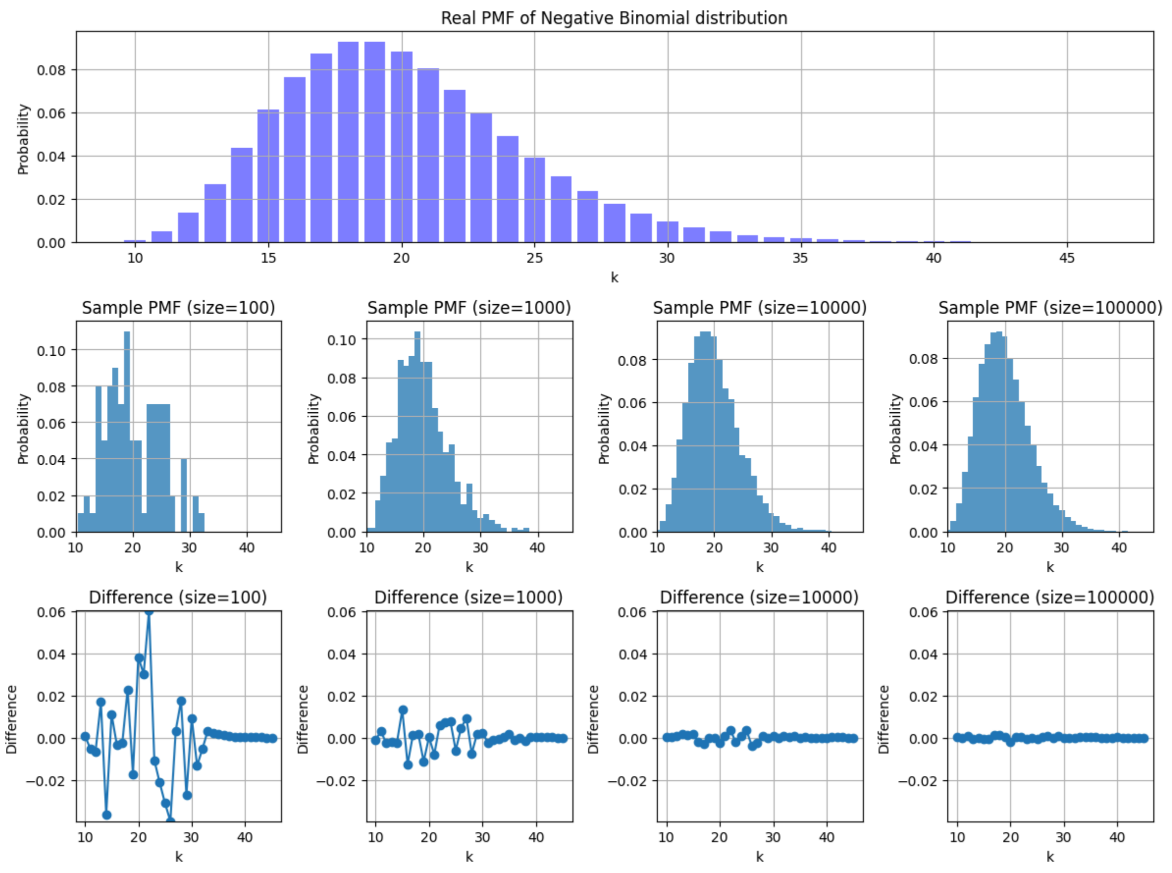
In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

def geometric_sample(p, size):
    u = np.random.rand(size)
    k = np.floor(np.log(u) / np.log(1 - p)) + 1
    return k.astype(int)

def geometric_pmf(k, p):
    return (1 - p)**(k - 1) * p
```

```python
# Parameters
p = 0.5
sizes = [10**2, 10**3, 10**4, 10**5]

# Generate samples
samples = {size: geometric_sample(p, size) for size in sizes}

# Plot real PMF
plt.figure(figsize=(12, 9))
k_max = max(max(samples[size]) for size in sizes)
k_values = np.arange(1, k_max + 1)
real_pmf = geometric_pmf(k_values, p)
plt.subplot(3, 1, 1)
plt.bar(k_values, real_pmf, alpha=0.5, color='blue', label='Real PMF')
plt.title('Real PMF of Geometric distribution')
plt.xlabel('k')
plt.ylabel('Probability')
plt.grid(True)

# Initialize variables to store maximum and minimum normalized differences
max_diff = float('-inf')
min_diff = float('inf')

# Plot sample PMFs and differences
for i, size in enumerate(sizes):
    plt.subplot(3, len(sizes), len(sizes) + i + 1)
    plt.hist(samples[size], bins=np.arange(k_max + 1) - 0.5, density=True, alpha
    plt.title(f'Sample PMF (size={size})')
    plt.xlabel('k')
    plt.ylabel('Probability')
    plt.grid(True)
    plt.xlim(0, k_max)

    # Plot the difference between real PMF and sample PMF
    plt.subplot(3, len(sizes), 2*len(sizes) + i + 1)
    sample_pmf, _ = np.histogram(samples[size], bins=np.arange(k_max + 1) - 0.5,
    difference = (real_pmf[:len(sample_pmf)] - sample_pmf) / np.sqrt(size)
    plt.plot(k_values[:len(sample_pmf)], difference, marker='o', linestyle='-')
    plt.title(f'Difference (size={size})')
    plt.xlabel('k')
    plt.ylabel('Normalized Difference')
    plt.grid(True)

    # Update maximum and minimum differences
    max_diff = max(max_diff, np.max(difference))
    min_diff = min(min_diff, np.min(difference))

# Set y-axis limits for all difference plots
for i in range(len(sizes)):
    plt.subplot(3, len(sizes), 2*len(sizes) + i + 1)
    plt.ylim(min_diff, max_diff)

plt.tight_layout()
plt.show()
```



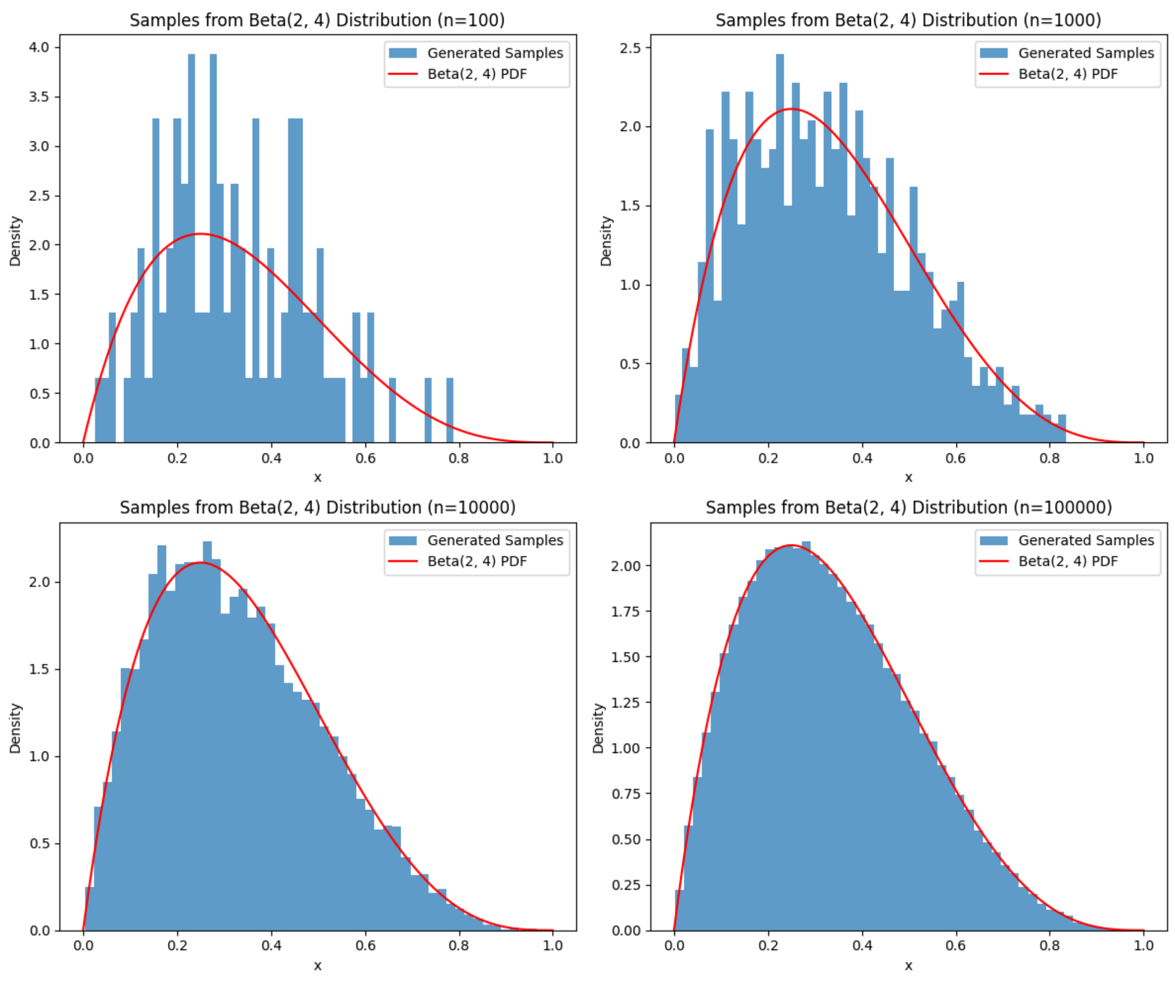
In [ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import nbinom

def negative_binomial_sample(r, p, size):
    return r + nbinom.ppf(np.random.rand(size), r, p)

def negative_binomial_pmf(k, r, p):
    return nbinom.pmf(k - r, r, p)

# Parameters
r = 10
p = 0.5
sizes = [10**2, 10**3, 10**4, 10**5]

# Generate samples
samples = {size: negative_binomial_sample(r, p, size) for size in sizes}

# Plot real PMF
plt.figure(figsize=(12, 9))
k_max = max(max(samples[size]) for size in sizes)
k_values = np.arange(r, k_max + 1)
real_pmf = negative_binomial_pmf(k_values, r, p)
plt.subplot(3, 1, 1)
plt.bar(k_values, real_pmf, alpha=0.5, color='blue', label='Real PMF')
plt.title('Real PMF of Negative Binomial distribution')
plt.xlabel('k')
plt.ylabel('Probability')
plt.grid(True)

# Initialize variables to store maximum and minimum normalized differences
max_diff = float('-inf')
min_diff = float('inf')
```

```
# Plot sample PMFs and differences
for i, size in enumerate(sizes):
    plt.subplot(3, len(sizes), len(sizes) + i + 1)
    plt.hist(samples[size], bins=np.arange(r, k_max + 1) - 0.5, density=True, al
    plt.title(f'Sample PMF (size={size})')
    plt.xlabel('k')
    plt.ylabel('Probability')
    plt.grid(True)
    plt.xlim(r, k_max)

    # Plot the difference between real PMF and sample PMF
    plt.subplot(3, len(sizes), 2*len(sizes) + i + 1)
    sample_pmf, _ = np.histogram(samples[size], bins=np.arange(r, k_max + 1) - 0
    difference = real_pmf[:len(sample_pmf)] - sample_pmf
    plt.plot(k_values[:len(sample_pmf)], difference, marker='o', linestyle='-')
    plt.title(f'Difference (size={size})')
    plt.xlabel('k')
    plt.ylabel('Difference')
    plt.grid(True)

    # Update maximum and minimum differences
    max_diff = max(max_diff, np.max(difference))
    min_diff = min(min_diff, np.min(difference))

# Set y-axis limits for all difference plots
for i in range(len(sizes)):
    plt.subplot(3, len(sizes), 2*len(sizes) + i + 1)
    plt.ylim(min_diff, max_diff)

plt.tight_layout()
plt.show()
```



Problem 3

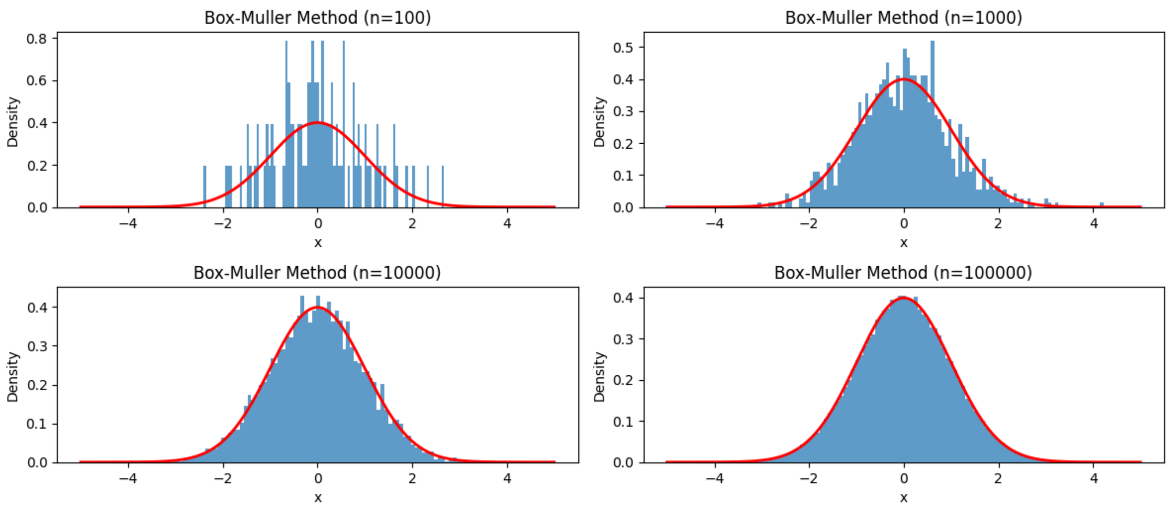
In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import beta

def target_pdf(x):
    """Calculate the PDF of the Beta(2, 4) distribution."""
    return beta.pdf(x, 2, 4)

def proposal_pdf(x):
    """Calculate the PDF of the uniform distribution U(0, 1)."""
    return 1

def generate_beta_samples(size):
    """Generate samples from the Beta(2, 4) distribution using the Acceptance-Re
    samples = []
    M = 10  # Choose a suitable constant M
    while len(samples) < size:
        x = np.random.uniform(0, 1)
        y = np.random.uniform(0, M)
        if y <= target_pdf(x) / proposal_pdf(x):
            samples.append(x)
    return samples

# Set sample sizes
sample_sizes = [int(1e2), int(1e3), int(1e4), int(1e5)]

# Create a 2x2 subplot grid
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Generate and plot samples for different sizes
for i, size in enumerate(sample_sizes):
    row = i // 2
    col = i % 2
    samples = generate_beta_samples(size)
    axes[row, col].hist(samples, bins=50, density=True, alpha=0.7, label='Genera
    x = np.linspace(0, 1, 100)
    axes[row, col].plot(x, target_pdf(x), 'r-', label='Beta(2, 4) PDF')
    axes[row, col].set_xlabel('x')
    axes[row, col].set_ylabel('Density')
    axes[row, col].set_title(f'Samples from Beta(2, 4) Distribution (n={size})')
    axes[row, col].legend()

# Adjust the spacing between subplots
plt.tight_layout()

plt.show()
```

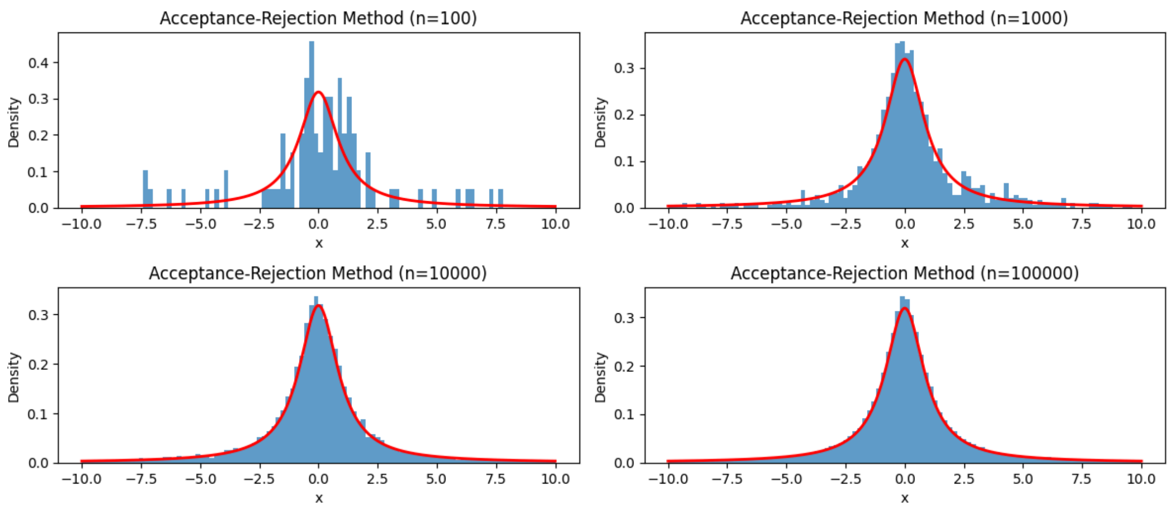


```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt

        def box_muller_samples(size):
            """Generate samples from the standard Normal distribution using the Box-Mull
            samples = []
            while len(samples) < size:
                u1 = np.random.uniform(0, 1)
                u2 = np.random.uniform(0, 1)
                z1 = np.sqrt(-2 * np.log(u1)) * np.cos(2 * np.pi * u2)
                z2 = np.sqrt(-2 * np.log(u1)) * np.sin(2 * np.pi * u2)
                samples.extend([z1, z2])
            return samples[:size]

        def acceptance_rejection_samples(size):
            """Generate samples from the standard Normal distribution using the Acceptan
            samples = []
            while len(samples) < size:
                x = np.random.standard_cauchy()
                y = np.random.uniform(0, 1)
                if y <= np.exp(-0.5 * x**2) / (np.sqrt(2*np.pi) * np.exp(-0.5 * x**2)):
                    samples.append(x)
            return samples

        # Set sample sizes
        sample_sizes = [int(1e2), int(1e3), int(1e4), int(1e5)]

        # Generate and plot samples for different sizes using Box-Muller method
        plt.figure(figsize=(12, 10))
        for i, size in enumerate(sample_sizes):
```

```
    samples = box_muller_samples(size)
    plt.subplot(4, 2, i+1)
    plt.hist(samples, bins=100, density=True, alpha=0.7)

    # Plot PMF curve
    x = np.linspace(-5, 5, 1000)
    pdf = 1 / np.sqrt(2 * np.pi) * np.exp(-0.5 * x**2)
    plt.plot(x, pdf, 'r', linewidth=2)

    plt.xlabel('x')
    plt.ylabel('Density')
    plt.title(f'Box-Muller Method (n={size})')
plt.tight_layout()

# Generate and plot samples for different sizes using Acceptance-Rejection metho
plt.figure(figsize=(12, 10))
for i, size in enumerate(sample_sizes):
    samples = acceptance_rejection_samples(size)
    plt.subplot(4, 2, i+1)
    plt.hist(samples, bins=100, density=True, range=(-10, 10), alpha=0.7)

    # Plot PMF curve
    x = np.linspace(-10, 10, 1000)
    pdf = 1 / (np.pi * (1 + x**2))
    plt.plot(x, pdf, 'r', linewidth=2)

    plt.xlabel('x')
    plt.ylabel('Density')
    plt.title(f'Acceptance-Rejection Method (n={size})')
plt.tight_layout()

plt.show()
```

```
C:\Users\pangfei\AppData\Local\Temp\ipykernel_27004\3419656836.py:21: RuntimeWarn
ing: invalid value encountered in double_scalars
  if y <= np.exp(-0.5 * x**2) / (np.sqrt(2*np.pi) * np.exp(-0.5 * x**2)):
```





Pros and cons:

Box-Muller:

- Pros: It is easy to implement, and the method only uses Unif(0, 1) as the basis data sample, which is simple to sample.
- Cons: Only the standard normal distribution can be sampled by this method.

Acceptance-Rejection:

- Pros: It can sample many kinds of probability distribution including many distributions that is difficult to sample directly.
- Cons: The domain of function g(x) must cover the domain of function f(x). If c is closed to 1, the basis distribution g is still difficult to sample; while if c is closed to 0, the probability of acceptance success will be small, which will cause low efficiency

Problem 4

(a) objective PDF: $f_{X,Y}(x,y) = \dfrac{1}{\pi \cdot a \cdot b}$

$x = \rho \cdot a \cdot cos\theta$, $y = \rho \cdot b \cdot sin\theta$, $\rho \in [0,1]$, $\theta \in [0,2\pi]$

$\theta(x,y) \in E_2(a,b)$

$$|J| = \left|\frac{\partial(x,y)}{\partial(\rho,\theta)}\right| = \left|\begin{bmatrix} \frac{\partial x}{\partial \rho} & \frac{\partial x}{\partial \theta} \\ \frac{\partial y}{\partial \rho} & \frac{\partial y}{\partial \theta} \end{bmatrix}\right| = \left|\begin{bmatrix} acos\theta & -\rho asin\theta \\ bsin\theta & \rho bcos\theta \end{bmatrix}\right| = \rho ab$$

$$\Rightarrow f_{\Rho,\Theta}(\rho,\theta) = f_{X,Y}(x,y) \cdot |J| = \frac{1}{\pi \cdot a \cdot b} \cdot \rho ab = \frac{\rho}{\pi}, \rho \in [0,1], \theta \in [0,2\pi]$$

$$\Rightarrow f_{\Rho}(\rho) = \int_0^{2\pi} \frac{\rho}{\pi} d\theta = 2\rho, 0 \le \rho \le 1$$

Therefore, CDF: $F_{\Rho}(\rho) = \rho^2, 0 \le \rho \le 1$

$$\Rightarrow f_\Theta(\theta) = \int_0^1 \frac{\rho}{\pi} d\rho = \frac{1}{2\pi}, 0 \le \theta \le 2\pi$$

Therefore, $\Theta \sim \mathrm{Unif}(0, 2\pi)$

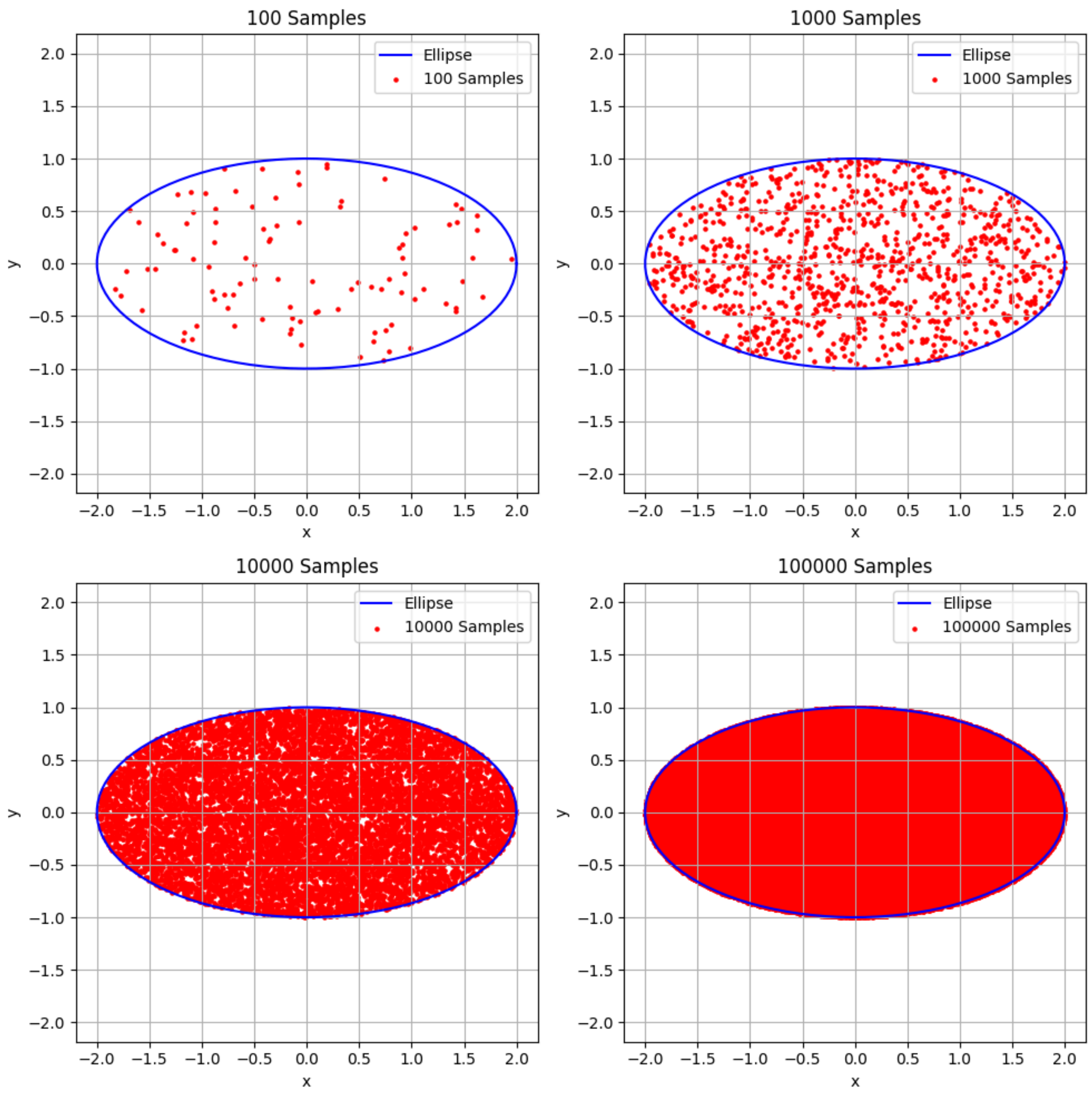
$\Rightarrow f_{\Rho,\Theta}(\rho, \theta) = f_{\Rho}(\rho) \cdot f_\Theta(\theta)$, $\Rho, \Theta$ are independent

$F^{-1}_{\Rho}(z) = \sqrt{z}, 0 \le z \le 1$

Let $U_1, U_2 \sim \mathrm{Unif}(0,1)$ and they are independent.

Therefore, $X \leftarrow a\sqrt{u_1}cos(2\pi u_2), Y \leftarrow b\sqrt{u_1}sin(2\pi u_2)$.

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

# Function to generate samples from the ellipse
def generate_samples(n_samples):
    # Generate random samples for U1 and U2
    U1 = np.random.rand(n_samples)
    U2 = np.random.rand(n_samples)

    # Transform U1 and U2 to x and y coordinates within the ellipse
    x = 2 * np.sqrt(U1) * np.cos(2 * np.pi * U2)
    y = np.sqrt(U1) * np.sin(2 * np.pi * U2)

    return x, y

# Generate samples for different numbers of samples
n_samples_list = [100, 1000, 10000, 100000]

# Create a 2x2 grid of plots
fig, axs = plt.subplots(2, 2, figsize=(10, 10))

# Iterate through each subplot and generate samples
for i, ax in enumerate(axs.flat):
    n_samples = n_samples_list[i]
    x_samples, y_samples = generate_samples(n_samples)

    # Plot the ellipse and the generated samples
    theta = np.linspace(0, 2 * np.pi, 100)
    ellipse_x = 2 * np.cos(theta)
    ellipse_y = np.sin(theta)
    ax.plot(ellipse_x, ellipse_y, color='blue', label='Ellipse')

    ax.scatter(x_samples, y_samples, color='red', s=5, label=f'{n_samples} Sampl
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title(f'{n_samples} Samples')
    ax.axis('equal')
    ax.legend()
    ax.grid(True)

plt.tight_layout()
plt.show()
```



(b) We easily obtain $f_{x,y,z}(x,y,z) = \frac{1}{4\pi r^2}$.

We have

$$\begin{cases} x = r\sin(\theta)\cos(\phi) \\ y = r\sin(\theta)\sin(\phi) \\ z = r\cos(\theta) \end{cases}$$

Jacobi Matrix M =

$$\begin{bmatrix} \frac{\partial x}{\partial \theta} & \frac{\partial x}{\partial \phi} \\ \frac{\partial y}{\partial \theta} & \frac{\partial y}{\partial \phi} \\ \frac{\partial z}{\partial \theta} & \frac{\partial z}{\partial \phi} \end{bmatrix}$$

=

$$\begin{bmatrix} r\cos\theta\cos\phi & -r\sin\theta\sin\phi \\ r\cos\theta\sin\phi & r\sin\theta\cos\phi \\ -r\sin\theta & 0 \end{bmatrix}$$

gram matrix $G = M^T M =$

$$\begin{bmatrix} r^2 & 0 \\ 0 & r^2 \sin^2\theta \end{bmatrix}$$

So $det(G) = r^4 \sin^2\theta$, $\sqrt{det(G)} = r^2 \sin\theta$.

$f_{\Theta,\Phi}(\theta,\phi) = f_{X,Y,Z}(x,y,z) \cdot \sqrt{det(G)} = \frac{1}{4\pi}\sin\theta$.

$f_\Theta(\theta) = \int_0^{2\pi} f_{\Theta,\Phi}(\theta,\phi)\,d\phi = \frac{1}{2}\sin\theta;\, 0 \le \theta \le \pi$.

$F_\Theta(\theta) = \int_0^\theta f_\Theta(s)\,ds = \frac{1-\cos\theta}{2}$.

$f_\Phi(\phi) = \frac{1}{2\pi}; 0 \le \phi \le 2\pi. \to f_{\Theta,\Phi}(\theta,\phi) = f_\Theta(\theta) \cdot f_\Phi(\phi)$.

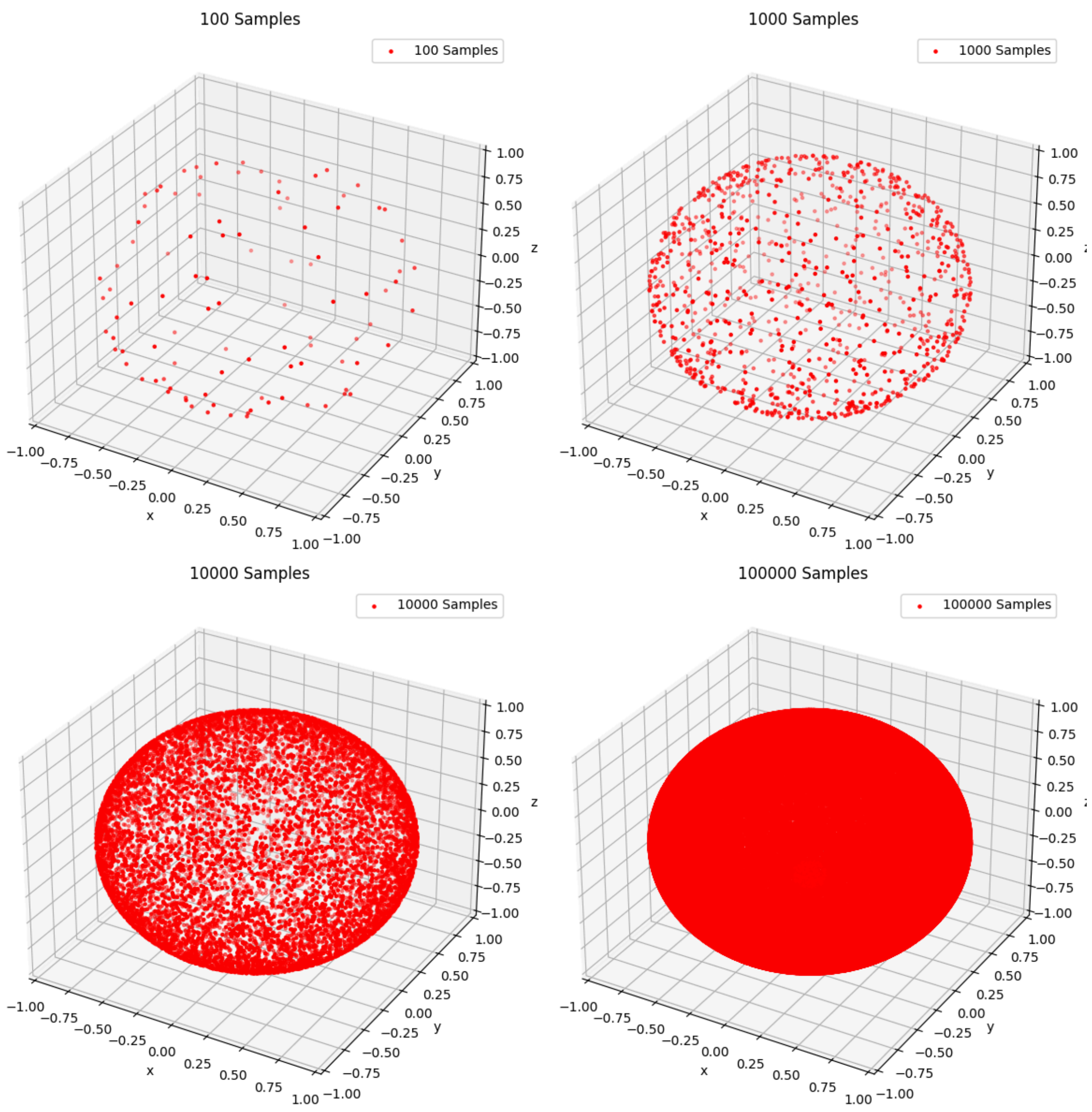
So θ and ϕ are independent; $\phi \sim Unif(0,2\pi) = 2\pi Unif(0,1)$.

So $F_\Theta^{-1}(s) = \arccos(1-2s);\ 0 \le s \le 1$.

Finally, we independently generate $U_1, U_2 \sim Unif(0,1)$.

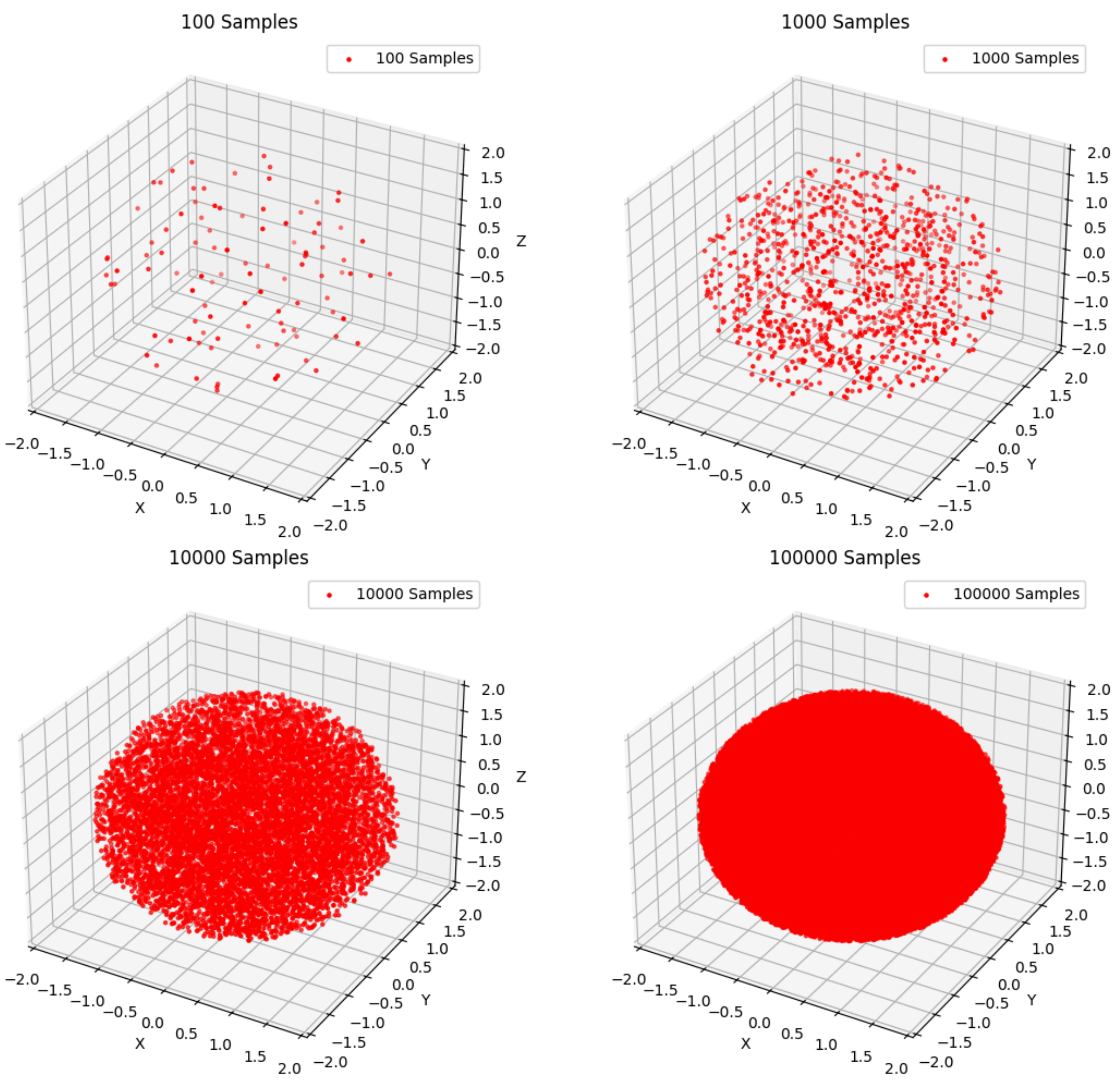
$$\begin{cases} X \leftarrow r \cdot 2\sqrt{U_1(1-U_1)} \cdot \cos(2\pi U_2) \\ Y \leftarrow r \cdot 2\sqrt{U_1(1-U_1)} \cdot \sin(2\pi U_2) \\ Z \leftarrow r \cdot (1-2U_1) \end{cases}$$

In [ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Function to generate samples from the sphere
def generate_samples(n_samples):
    # Generate random samples for U1 and U2
    U1 = np.random.rand(n_samples)
    U2 = np.random.rand(n_samples)

    # Transform U1 and U2 to x, y, and z coordinates on the sphere
    x = 2 * np.sqrt(U1 * (1 - U1)) * np.cos(2 * np.pi * U2)
    y = 2 * np.sqrt(U1 * (1 - U1)) * np.sin(2 * np.pi * U2)
    z = 1 - 2 * U1

    return x, y, z

# Generate samples for different numbers of samples
n_samples_list = [100, 1000, 10000, 100000]

# Create a 2x2 grid of plots
fig, axs = plt.subplots(2, 2, figsize=(12, 12), subplot_kw={'projection': '3d'})

# Iterate through each subplot and generate samples
for i, ax in enumerate(axs.flat):
    n_samples = n_samples_list[i]
    x_samples, y_samples, z_samples = generate_samples(n_samples)

    # Plot the sphere and the generated samples
    ax.scatter(x_samples, y_samples, z_samples, color='red', s=5, label=f'{n_sam
    ax.set_title(f'{n_samples} Samples')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.set_xlim(-1, 1)
    ax.set_ylim(-1, 1)
    ax.set_zlim(-1, 1)
    ax.legend()

plt.tight_layout()
plt.show()
```



(c) We use acceptance-rejection method. First, we randomly generalize points in a 2x2x2 box, then reject those d > 2.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def sample_from_cube(n_samples):
    # Generate n_samples points uniformly within a cube [-2, 2] x [-2, 2] x [-2,
    x = np.random.uniform(-2, 2, size=n_samples)
    y = np.random.uniform(-2, 2, size=n_samples)
    z = np.random.uniform(-2, 2, size=n_samples)
    return np.column_stack((x, y, z))

def acceptance_rejection_sampling(n_samples, r=2):
    samples = []
    while len(samples) < n_samples:
        # Sample a point from the cube
        sample = sample_from_cube(1)[0]
        # Check if the point is inside the sphere
        if np.linalg.norm(sample) <= r:
            samples.append(sample)
    return np.array(samples)

# Sample sizes
sample_sizes = [100, 1000, 10000, 100000]

# Create 2x2 subplot grid
fig, axs = plt.subplots(2, 2, figsize=(12, 10), subplot_kw={'projection': '3d'})

# Generate samples and plot for each subplot
for i, ax in enumerate(axs.flat):
    # Generate samples
    samples = acceptance_rejection_sampling(sample_sizes[i])
    # Plot samples
    ax.scatter(samples[:, 0], samples[:, 1], samples[:, 2], c='r', s=5, label=f'
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title(f'{sample_sizes[i]} Samples')
    ax.set_xlim(-2, 2)
    ax.set_ylim(-2, 2)
    ax.set_zlim(-2, 2)
    ax.legend()

plt.tight_layout()
plt.show()
```



Problem 5

The probability of landing in the betman sign = The area of betman sign / The sample space

The upper bound and lower bound are displayed the code below:

In [ ]:
```
import numpy as np

def f(x):
    return 4 / (1 + x**2)

def monte_carlo_integration(n_samples):
    # Generate random x values
    x_values = np.random.rand(n_samples)

    # Evaluate function values
    function_values = f(x_values)

    # Compute the average function value
    average_function_value = np.mean(function_values)

    # Estimate the integral
    integral_estimate = average_function_value * 1  # Interval width is 1

    return integral_estimate

# Number of samples for Monte Carlo integration
n_samples_list = [10**3, 10**4, 10**5, 10**6]

# True value of pi
true_pi = np.pi

# Perform Monte Carlo integration for different numbers of samples
for n_samples in n_samples_list:
    integral_estimate = monte_carlo_integration(n_samples)
    difference_with_pi = np.abs(integral_estimate - true_pi)
    print(f"Number of samples: {n_samples}, Monte Carlo Integral Estimate: {inte
```

```
Number of samples: 1000, Monte Carlo Integral Estimate: 3.153527276295453, Differ
ence with Pi: 0.011934622705660036
Number of samples: 10000, Monte Carlo Integral Estimate: 3.144299571394213, Diffe
rence with Pi: 0.002706917804419895
Number of samples: 100000, Monte Carlo Integral Estimate: 3.143433664455691, Diff
erence with Pi: 0.0018410108658977187
Number of samples: 1000000, Monte Carlo Integral Estimate: 3.1420836815656052, Di
fference with Pi: 0.0004910279758121305
```

In [ ]:

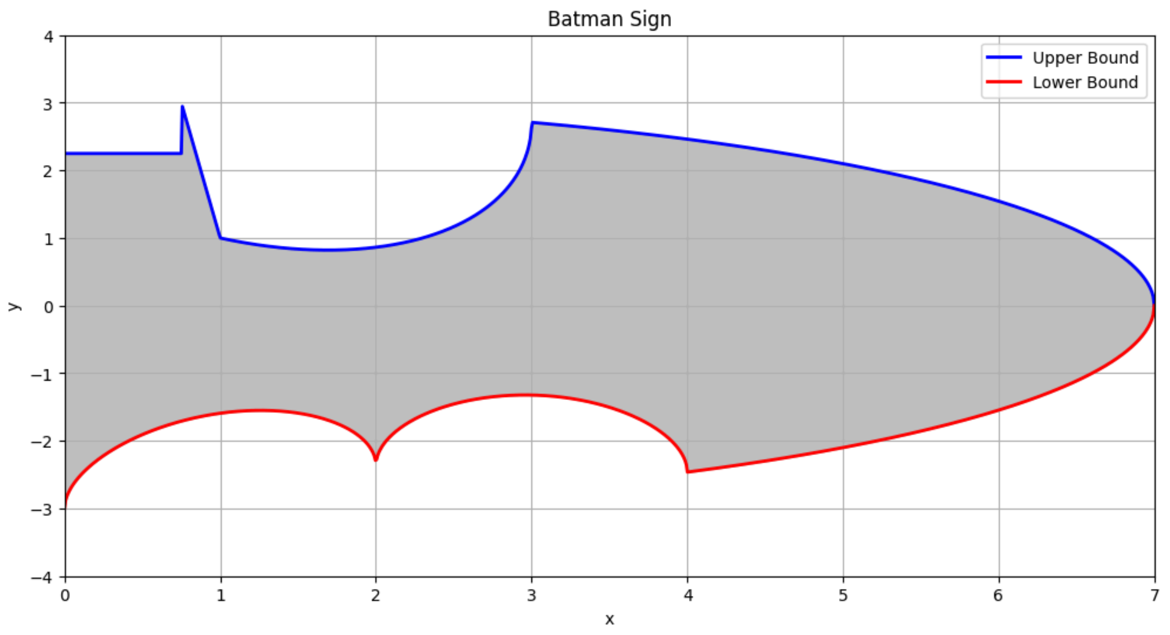
```
import numpy as np

def f(x):
    return np.sqrt(x + np.sqrt(x + np.sqrt(x + np.sqrt(x))))

def monte_carlo_integration(n_samples):
    # Generate random x values
    x_values = np.random.uniform(0, 4, n_samples)

    # Evaluate function values
    function_values = f(x_values)

    # Compute the average function value
    average_function_value = np.mean(function_values)

    # Estimate the integral
    integral_estimate = average_function_value * 4  # Interval width is 4

    return integral_estimate

# Ground truth value
ground_truth = 7.6766100019

# Number of samples for Monte Carlo integration
n_samples_list = [10**3, 10**4, 10**5, 10**6]

# Perform Monte Carlo integration for different numbers of samples
for n_samples in n_samples_list:
    integral_estimate = monte_carlo_integration(n_samples)
    difference_with_ground_truth = np.abs(integral_estimate - ground_truth)
    print(f"Number of samples: {n_samples}, Monte Carlo Integral Estimate: {inte
Number of samples: 1000, Monte Carlo Integral Estimate: 7.586002240120432, Differ
ence with Ground Truth: 0.09060776177956864
Number of samples: 10000, Monte Carlo Integral Estimate: 7.6707731205402405, Diff
erence with Ground Truth: 0.0058368813597597935
Number of samples: 100000, Monte Carlo Integral Estimate: 7.69213645506796, Diffe
rence with Ground Truth: 0.01552645316795953
Number of samples: 1000000, Monte Carlo Integral Estimate: 7.67673582788308, Diff
erence with Ground Truth: 0.0001258259830798636
```

In [ ]:
```python
import numpy as np
from scipy.stats import norm, t

# Define the target distribution (standard normal)
target_distribution = norm(loc=0, scale=1)

# Define the importance sampling distribution (Student's t-distribution)
# You can adjust the degrees of freedom parameter to control the tail behavior
importance_sampling_distribution = t(df=3)

# Number of samples for importance sampling
n_samples = 10000000

# Generate random samples from the importance sampling distribution
importance_samples = importance_sampling_distribution.rvs(size=n_samples)

# Compute importance sampling weights
target_pdf_values = target_distribution.pdf(importance_samples)
importance_sampling_pdf_values = importance_sampling_distribution.pdf(importance
importance_sampling_weights = target_pdf_values / importance_sampling_pdf_values

# Estimate the probability of the rare event
c_estimate = np.mean(importance_sampling_weights * (importance_samples > 8))

print("Estimated probability of the rare event:", c_estimate)
```

```
Estimated probability of the rare event: 5.931657023630913e-16
```

Problem 6

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

def upper_bound(x):
    """Upper bound function for Batman sign."""
    y = np.zeros_like(x)
    mask1 = (0 <= x) & (x < 3/4)
    mask2 = (3/4 <= x) & (x < 1)
    mask3 = (1 <= x) & (x < 3)
    mask4 = (3 <= x) & (x <= 7)

    y[mask1] = 9/4
    y[mask2] = -8*x[mask2] + 9
    y[mask3] = 6 * np.sqrt(10)/7 + 3/2 - x[mask3]/2 - 3*np.sqrt(10)*np.sqrt(4 -
    y[mask4] = 3*np.sqrt(1 - x[mask4]**2/49)

    return y

def lower_bound(x):
    """Lower bound function for Batman sign."""
    y = np.zeros_like(x)
    mask1 = (0 <= x) & (x < 4)
    mask2 = (4 <= x) & (x <= 7)

    y[mask1] = x[mask1]/2 + np.sqrt(1 - (np.abs(x[mask1]-2)-1)**2) - ((3*np.sqrt
    y[mask2] = -3*np.sqrt(1 - x[mask2]**2/49)

    return y

# Define the range of x values
x = np.linspace(0, 7, 1000)

# Upper bound function values
y_upper = upper_bound(x)

# Lower bound function values
y_lower = lower_bound(x)

# Plot the Batman sign
plt.figure(figsize=(12, 6))
plt.plot(x, y_upper, 'b', linewidth=2, label='Upper Bound')
plt.plot(x, y_lower, 'r', linewidth=2, label='Lower Bound')
plt.fill_between(x, y_lower, y_upper, where=(y_lower <= y_upper), color='gray',
plt.xlabel('x')
plt.ylabel('y')
plt.title('Batman Sign')
plt.legend()
plt.grid(True)
plt.xlim(0, 7)
plt.ylim(-4, 4)
plt.show()

# Estimate the area using Monte Carlo method
num_samples = int(1e6)
x_samples = np.random.uniform(0, 7, num_samples)
y_samples = np.random.uniform(-4, 4, num_samples)
samples_within_bounds = (y_samples >= lower_bound(x_samples)) & (y_samples <= up
area_estimate = np.sum(samples_within_bounds) / num_samples * 7 * 8

print('Estimated area of 1/2 Batman sign:', area_estimate)
print('Estimated area of Batman sign:', 2*area_estimate)
```



```
Estimated area of 1/2 Batman sign: 24.140592
Estimated area of Batman sign: 48.281184
```