A Report on Assignment

Question 1:

Evaluate the methods listed above from sampling from the given function. Which of them is most algorithmically efficient? Which is most space efficient? Which is most computationally efficient?

Answer: Consider the function `target_distribution` in **Listing 1** as Modified Gaussian Probability Distribution p(x) for evaluating Rejection Sampling, MCMC-Metropolis-Hastings Sampling, MCMC-Gibbs Sampling and Inverse Transform Sampling given in subsequently.

Target Distribution p(x)

```
# Calculate function L(x)
def calculate_L(x, alpha, beta):
    return np.where(x < alpha, 0, beta * (x - 1))

# Calculate the exponential term modified by L(x)
def calculate_exp_term(x, mu, sigma, alpha, beta):
    return np.exp(-((x - mu) ** 2) / (2 * sigma ** 2)) -
    calculate_L(x, alpha, beta)

# Calculate the normalization constant Z
def calculate_constant_Z(mu, sigma, alpha, beta):
    z, error = quad(calculate_exp_term, -np.inf, np.inf, args=(mu,
    sigma, alpha, beta))
    return z

# Target distribution function (normalized)
def target_distribution(x, mu, sigma, alpha, beta, z):
    return calculate_exp_term(x, mu, sigma, alpha, beta) / z
```

Listing 1: Implementation of p(x)

Rejection Sampling

```python
# Rejection Sampling
def accept_reject_sample(mu, sigma, alpha, beta, z, max_iterations=1000):
    for _ in range(max_iterations):
        x = np.random.normal(mu, sigma)
        p_x = target_distribution(x, mu, sigma, alpha, beta, z)
        u = np.random.uniform(0, 1)
        if p_x > 0 and u <= p_x:
            return x
    raise ValueError("Failed to generate a valid sample from the distribution after max iterations.")

def rejection_sampling(mu, sigma, alpha, beta, num_samples, max_iterations=1000):
    x_samples = []
    z = calculate_constant_Z(mu, sigma, alpha, beta)
    while len(x_samples) < num_samples:
        x = accept_reject_sample(mu, sigma, alpha, beta, z)
        x_samples.append(x)
    return np.array(x_samples)
```

Listing 2: Implementation of Rejection Sampling

Metropolis-Hastings Sampling

```python
# Metropolis-Hastings Sampling
def metropolis_hastings_sampling(mu, sigma, alpha, beta, num_samples, proposal_std):
    x_samples = []
    z = calculate_constant_Z(mu, sigma, alpha, beta)
    x_current = np.random.normal(mu, sigma)
    while len(x_samples) < num_samples:
        x_proposal = np.random.normal(x_current, proposal_std)
        p_current = target_distribution(x_current, mu, sigma, alpha, beta, z)
        p_proposal = target_distribution(x_proposal, mu, sigma, alpha, beta, z)
        acceptance_ratio = min(1, p_proposal / p_current)
        if np.random.uniform(0, 1) < acceptance_ratio:
            x_current = x_proposal
        x_samples.append(x_current)
    return np.array(x_samples)
```

Listing 3: Implementation of Metropolis-Hastings Sampling

Gibbs Sampling

```python
# Gibbs Sampling
def generate_sample_g_y_given_x(x):
    mean_y = x
    sigma_y = 1
    return np.random.normal(mean_y, sigma_y)

def update_parameters(x, y, alpha, beta, mu, sigma):
    alpha = np.mean(x) + 0.1
    beta = np.std(x)
    mu = np.mean(y)
    sigma = np.std(y) + 1
    return alpha, beta, mu, sigma

def gibbs_sampling(mu, sigma, alpha, beta, num_samples,
    max_iterations=1000):
    x_samples = []
    y_samples = []
    z = calculate_constant_Z(mu, sigma, alpha, beta)
    while len(x_samples) < num_samples:
        xi = accept_reject_sample(mu, sigma, alpha, beta, z)
        yi = generate_sample_g_y_given_x(xi)
        alpha, beta, mu, sigma = update_parameters([xi], [yi],
    alpha, beta, mu, sigma)
        x_samples.append(xi)
        y_samples.append(yi)
    return np.array(x_samples)
```

Listing 4: Implementation of Gibbs Sampling

Inverse Transform Sampling

```python
# Inverse Transform Sampling
def inverse_transform_sampling(mu, sigma, alpha, beta, num_samples)
    :
    z = calculate_constant_Z(mu, sigma, alpha, beta)
    x_values = np.linspace(-10, 10, 1000)
    pdf_values = target_distribution(x_values, mu, sigma, alpha,
    beta, z)
    cdf_values = cumulative_trapezoid(pdf_values, x_values, initial
    =0)
    cdf_values /= cdf_values[-1]
    inverse_cdf = interp1d(cdf_values, x_values, bounds_error=False
    , fill_value="extrapolate")
    uniform_samples = np.random.uniform(0, 1, num_samples)
    x_samples = inverse_cdf(uniform_samples)
    return np.array(x_samples)
```

Listing 5: Implementation of Gibbs Sampling

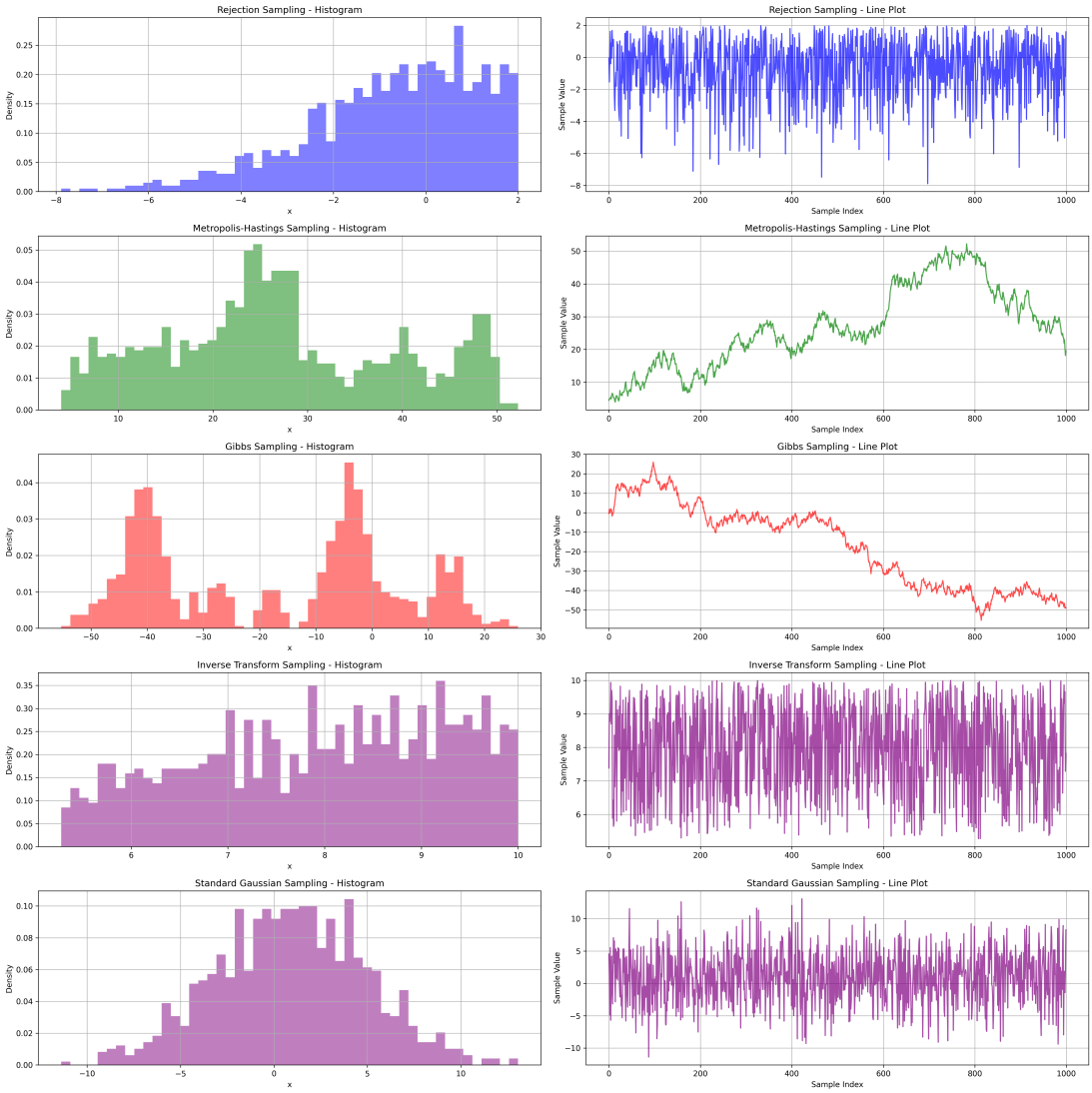

Figure 1: Sampling Methods Comparison

Sampling Methods Comparison

A program (file name **Comparison_Sampling_Methods.py**) for those sampling methods is implemented in **Python** and it also used standard library numpy and scipy. The method quad is used for integral function for calculating constant z. The Figure 1 shows a comparison of sampling among different methods implemented. The runtimes of those methods implemented shown in Figure 2.

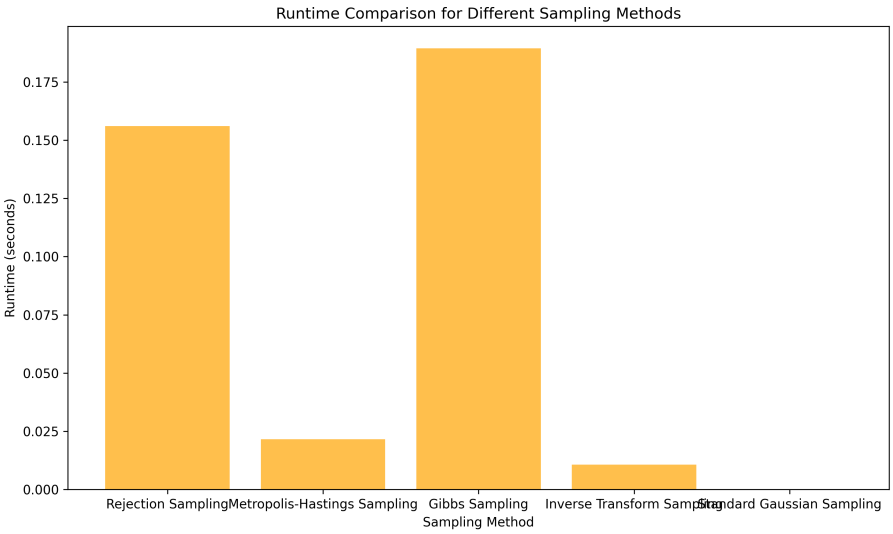

Figure 2: Runtime Comparison

Evaluation of Sampling Methods for Modified Gaussian Probability Distribution p(x)

The Most Algorithmic Efficiency:

- **Inverse Transform Sampling** is the most efficient in terms of directness and speed for distributions with known CDFs [1]

  The implementation uses the interp1d function from scipy.interpolate to create a continuous approximation of the inverse CDF from precomputed CDF values. This means that each sample can be directly obtained by a quick look-up and transformation, minimizing the number of steps needed to generate each sample. The process is deterministic. it does not involve trial and error for each sample results in eliminating the possibility of having a high rejection rate, which can be a bottleneck in algorithmic efficiency.

The Most Space Efficiency:

- **Inverse Transform Sampling** has the least memory requirements once the inverse CDF is known, making it the most space-efficient method. It only needs to store the precomputed CDF and inverse CDF. No iterative or rejection-related overhead. When memory resources are limited and when generating a large number of samples. This method gives advantage.

The Computational Efficiency:

- **Inverse Transform Sampling** also leads in computational efficiency for distributions with easily computed inverse CDFs. It has direct and fast transformation from uniform to target distribution. It shows a constant time complexity per sample generation by avoiding of iterative and dependent sampling steps. There is no elimination of rejection and acceptance overhead.

Question 2:

Write code using one of the methods that you outlined in Question 1 to sample from p(x). This can be done in whatever language you choose, using any libraries you choose.

Answer: The program can be found in file **Comparison_Sampling_Methods.py**

Question 3:

Analyze the complexity (in time, space, clock cycles, statistical efficiency) of your implementation. Compare this to a standard gaussian implementation and mention how you might improve your efficiency if given more time.

Answer:

For Inverse Transformation:

- Runtime: 0.01064 seconds. A runtime comparison is also given in Figure 2
- Memory: Sample size = 1000, Size of float 8 bytes (64-bit floating-point). For CDF, Inverse CDF, Uniform Samples and Generated Samples

$$4 * (1000) * 8bytes = 32MB$$

  The memory usage for Inverse Transform Sampling (ITS) with the parameters (CDF, Inverse CDF, Uniform Samples and Generated Samples) is 32 KB.

- Statistical Efficiency: 7.94424

Standard Gaussian Sampling:

- Runtime: 0.00003 seconds. A runtime comparison is also given in Figure 2
- Memory: Sample size = 1000

$$1 * (1000) * 8bytes = 8MB$$

- Statistical Efficiency: 0.92209

Question 4:

If you needed to implement this sampling on a specialized piece of hardware (GPU, FPGA, TPU, ASIC, etc.), would you change your answer for which sampling algorithm would be most efficient? How would your implementation change?

Answer: The choice is Inverse Transform Sampling. Compute the inverse CDF can be handled efficiently on GPUs using interpolation techniques as GPUs excel at performing interpolation and linear algebra operations. **cupy** library can be used.

Once the CDF and its inverse are computed, generating samples involves mapping uniform random numbers to the target distribution and this mapping is a highly parallelizable operation because each sample generation is independent of others. Morver, the algorithm involves reading from precomputed tables (CDF and inverse CDF), which can fit well with the memory access patterns for GPUs. [1]

My Comments

To do this assignment, I need significant academic research on Statistics as I have limited bandwith with statistics. However, due to limited time I had to rely on ChatGPT [1] for quick answers that may have errors or could be misleading. Therefore it requires further research and analysis. If I could discuss with expert in statistics, define the problem well and 100% understand I could do better. This industrial problem may involve in a quite R&D work when implementing with latest computing systems for both CPU and GPU.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. https://arxiv.org/abs/2005.14165, 2020.