Stochastic Simulation Coursework

Tarun Mistry CID: 02037785

Question 1

Proposal for the Banana Density Function

In order to efficiently sample from the Banana density, we need to select a proposal distribution

q(x) that not only captures the non-linear dependency between x_2 and x_1, but also provides a suitable spread to maximize the acceptance rate.

The proposed distribution should account for the complex relationship between the two variables, ensuring that the sampling process remains effective.

The Banana density function is given by:

$$p(x) = \exp\left(-\frac{x_1^2}{10} - \frac{x_2^2}{10} - 2\left(x_2 - x_1^2\right)^2\right)$$

This density function exhibits a non-linear structure in which x_2 is highly dependent on x_1. To model this, we can use the product of two Gaussian distributions:

- $x_1 \sim \mathcal{N}(0, 5),$ which provides an appropriate spread for x_1 centered around zero, n

- $x_2|x_1 \sim \mathcal{N}(x_1^2, \left(\frac{1}{2}\right)^2),$ ensuring that x_2 is centered around $x_1^2,$ matching the term

Thus, the proposal distribution can be written as:

$$q(x) = \frac{1}{\sqrt{2\pi \cdot 5}}\exp\left(-\frac{x_1^2}{2 \cdot 5}\right) \cdot \frac{1}{\sqrt{2\pi \cdot \left(\frac{1}{2}\right)^2}}\exp\left(-\frac{(x_2 - x_1^2)^2}{2 \cdot \left(\frac{1}{2}\right)^2}\right)$$

Simplifying this expression, we obtain:

$$q(x) = \frac{1}{2\pi\sqrt{5} \cdot \frac{1}{2}}\exp\left(-\frac{x_1^2}{2 \cdot 5} - \frac{(x_2 - x_1^2)^2}{2 \cdot \left(\frac{1}{2}\right)^2}\right)$$

Finding M

$$
\begin{aligned}
M &= \sup_{(x_1,x_2)} \frac{\bar{p}(x_1,x_2)}{q(x_1,x_2)} \\
&= \left(2\pi\sqrt{5}\cdot\frac{1}{2}\right) e^{\left(-\frac{x_1^2}{10}-\frac{x_2^2}{10}-2(x_2-x_1^2)^2+\frac{x_1^2}{2\cdot 5}+\frac{(x_2-x_1^2)^2}{2\cdot(\frac{1}{2})^2}\right)} \\
&= \pi\sqrt{5}e^{-\frac{x_2^2}{10}}, \\
\log M &= -\frac{x_2^2}{10} + \log(\pi\sqrt{5}), \\
\frac{d}{dx_2}(\log M) &= -\frac{x_2}{5} = 0, \\
&\text{Thus implying } x_2 = 0.
\end{aligned}
$$

So here is it clear by our choice of the variances and looking at the derivative of log(M) wrt x_2 that the supremum of M is found when x_2=0. From here we can proceed with a grid search to find this exact value then we can later verify with a numerical implementation by bounding $e^{-}(\cdot)$

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
import random
from scipy import stats, integrate
from scipy.special import gammaln
# np.random.default_rng(1)
```

Out[ ]:
```
Generator(PCG64) at 0x1A32CB29620
```

In [ ]:
```
# Defining the Banana density function
def banana_density(x_1, x_2):
    return np.exp(-x_1**2 / 10 - x_2**2 / 10 - 2 * (x_2 - x_1**2)**2)

# Then defining our proposal density q(x)
def proposal_density(x_1, x_2, s):
    return (2*math.pi*s[0]*s[1])**-1 * np.exp((-x_1**2)/(2*s[0]**2) - (x_2 - (x_

# conducting the sample
def sample_proposal(s):
    x_1 = np.random.normal(0, s[0])
    x_2 = np.random.normal(x_1**2, s[1])
    return x_1, x_2

# This is a function which calculates M by finding the maximum of p(x) / q(x)
def calculate_M(s):
    max_ratio = 0
    x1_vals = np.linspace(-10, 10, 1000)
    for xi in x1_vals:
        p_x = banana_density(xi, 0)
        q_x = proposal_density(xi, 0, s=s)
        if q_x > 0:
            ratio = p_x / q_x
            if ratio > max_ratio:
```

```
                max_ratio = ratio
    return max_ratio
```

In [ ]:
```
# Proceed with rejection sampling function
def rejection_sampler(N, s):
    M = calculate_M(s)
    total_samples = 0
    accepted_samples = []


    while len(accepted_samples) < N:
        # Now sampling from the proposal distribution q(x)
        x_1, x_2 = sample_proposal(s=s)
        p_x = banana_density(x_1, x_2)
        q_x = proposal_density(x_1, x_2, s=s)

        # Acceptance condition in log-domain and print(p_x, q_x)
        if p_x <= 0:
            continue
        if math.log(np.random.rand()) < (math.log(p_x) - math.log(q_x) - math.lo
            accepted_samples.append([x_1, x_2])

        total_samples += 1

    acceptance_rate = N / total_samples
    return np.array(accepted_samples), M, acceptance_rate
```

In [ ]:
```
# Run the rejection sampler and plot the results
N = 50000
samples, m, acceptance_rate = rejection_sampler(N, s=[math.sqrt(5), 0.5])
print(f"         M value: {m} \n Acceptance rate: {acceptance_rate}")
```

```
        M value: 7.025634452704435
Acceptance rate: 0.5096372402124169
```

We can further support this M value justification from the grid search via the analytical computation below

$$M = \sup_{(x_1,x_2)} \frac{\bar{p}(x_1,x_2)}{q(x_1,x_2)} \tag{1}$$

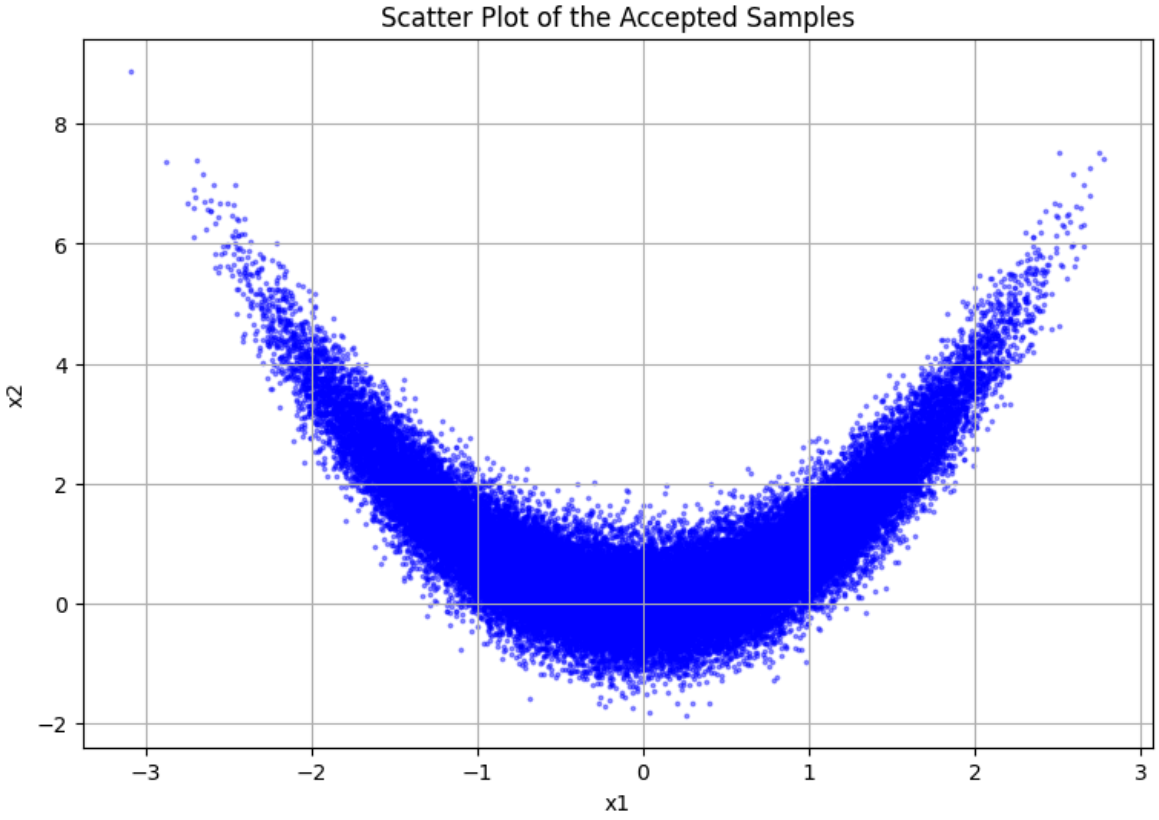
$$= \sup_{(x_1,x_2)} \left(2\pi\sqrt{5}\cdot\frac{1}{2}\right) \frac{\exp\left\{\cancel{-\frac{x_1^2}{10}} - \frac{x_2^2}{10} - \cancel{2(x_2-x_1^2)^2}\right\}}{\exp\left\{\cancel{-\frac{x_1^2}{10}} - \cancel{2(x_2-x_1^2)^2}\right\}} \tag{2}$$

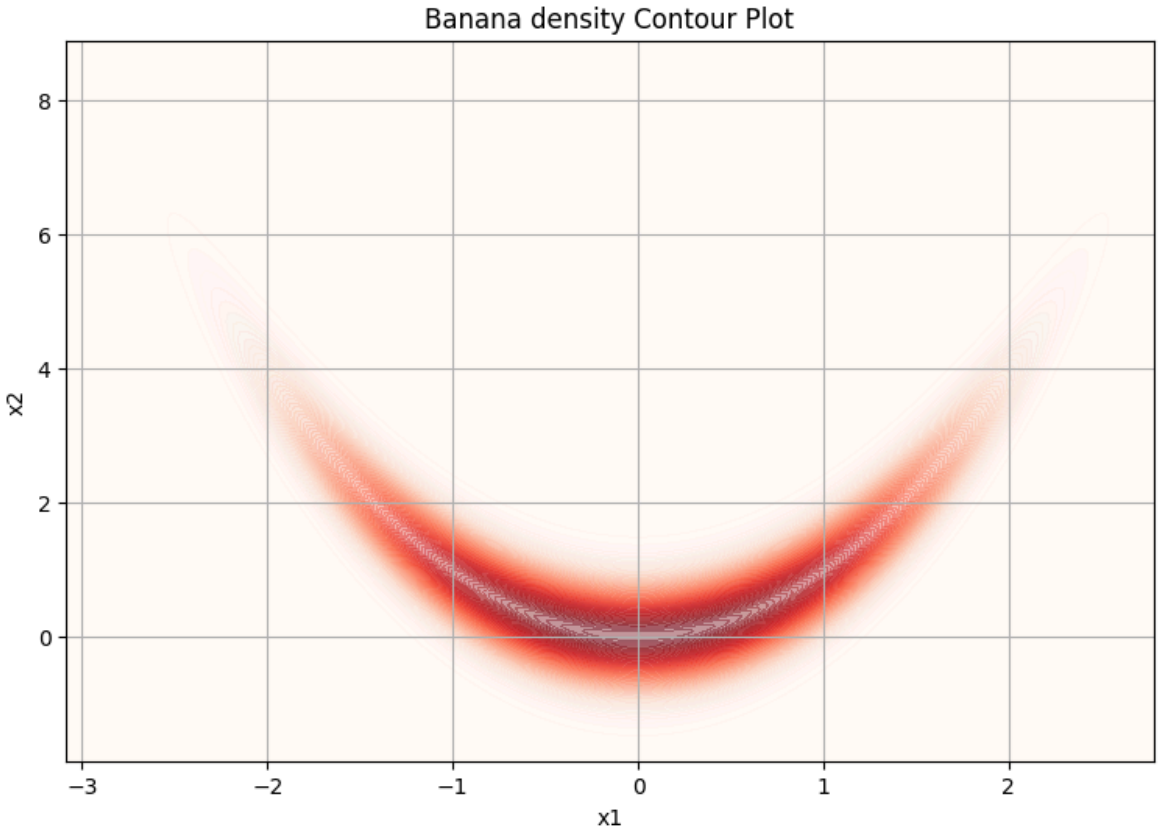
$$= \sup_{(x_1,x_2)} \pi\sqrt{5}\exp\left\{-\frac{x_2^2}{10}\right\} \tag{3}$$

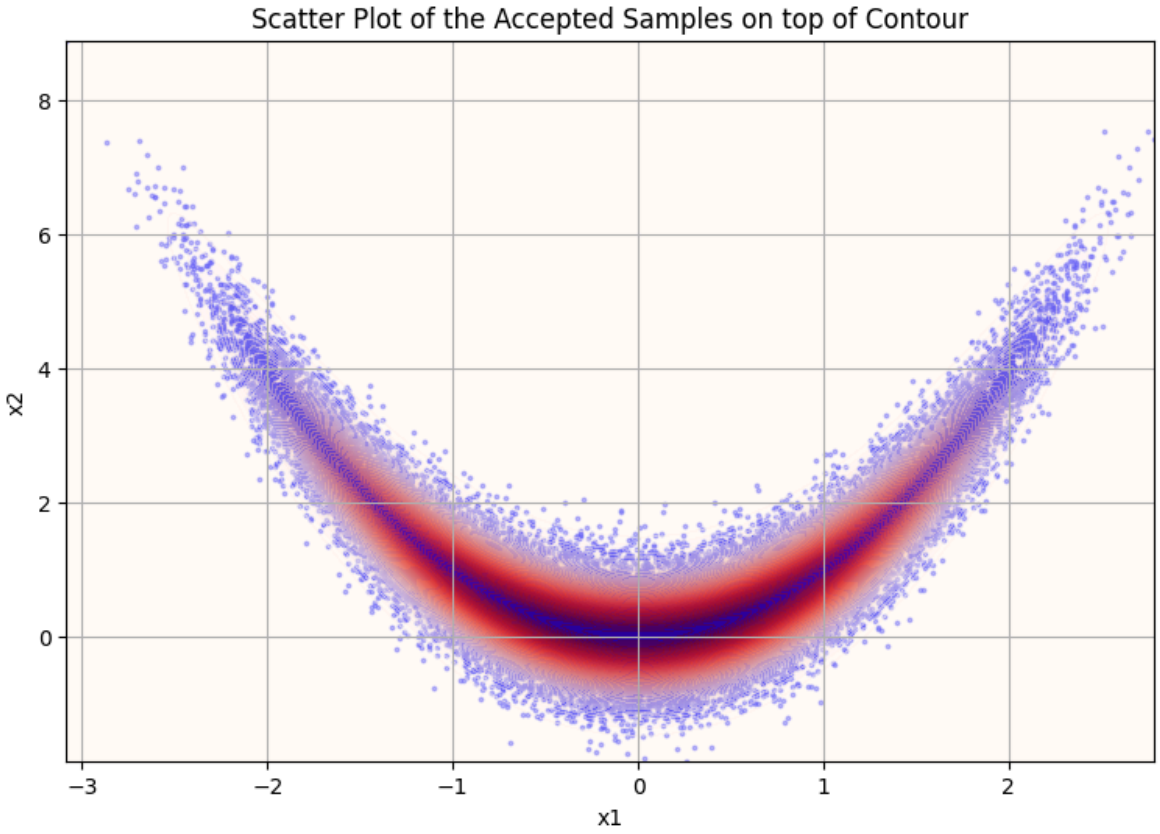
This is clearly maximised at $x_2 = 0 \implies M = \pi\sqrt{5} \approx 7.0256$ (as per grid search).

In [ ]:
```
#Creating the grid
x1_vals = np.linspace(min(samples[:, 0]), max(samples[:, 0]), 300)
x2_vals = np.linspace(min(samples[:, 1]), max(samples[:, 1]), 300)
X1, X2 = np.meshgrid(x1_vals, x2_vals)

#Defining the Banana density
Z = banana_density(X1, X2)
```

```
# This is the scatter plot (Plot 1)
plt.figure(figsize=(9, 6))
plt.scatter(samples[:, 0], samples[:, 1], s=3, alpha=0.4, label="Accepted Sample
plt.title("Scatter Plot of the Accepted Samples")
plt.xlabel("x1")
plt.ylabel("x2")
plt.grid()
plt.show()

# This is the contour plot (Plot 2)
plt.figure(figsize=(9, 6))
plt.contourf(X1, X2, Z, levels=100, cmap='Reds', alpha=.4)
plt.title("Banana density Contour Plot")
plt.xlabel("x1")
plt.ylabel("x2")
plt.grid()
plt.show()

# This is the scatter plot overlaid on the contour of the banana density (Plot 3
plt.figure(figsize=(9, 6))
plt.scatter(samples[:, 0], samples[:, 1], s=3, alpha=0.4, label="Accepted Sample
plt.contourf(X1, X2, Z, levels=100, cmap='Reds', alpha=.4)
plt.title("Scatter Plot of the Accepted Samples on top of Contour")
plt.xlabel("x1")
plt.ylabel("x2")
plt.grid()
plt.show()
```







1.4 Numerically approximating the normalisation constant

In [ ]:
```
def integrate_function(f, a, b):
    return integrate.dblquad(f, *a, *b)
z, e = integrate_function(banana_density, a=(-12, 12), b=(-12 ,12))
print(f'The integral of p(x) is {z:.6f}')
```

```
The integral of p(x) is 3.601326
```

In [ ]:
```
print(f'The theoretical acceptance rate comes to {float(z/m):.6f}')
```

```
The theoretical acceptance rate comes to 0.512598
```

Note how the theoretical acceptance rate (approximately 0.5126) aligns closely with the observed acceptance rate (approximately 0.5115), suggesting that the proposal density and sampling process have been implemented correctly. Across multiple iterations, the observed acceptance rate fluctuates near the theoretical value, occasionally exceeding or falling below it. This variability is expected given the inherent randomness in the sampling algorithm. Possibly, if allowed more computation time the empirical approximate would approach the theoretical value for α (The acceptance rate).

1.5

In [ ]:
```
banana_samples = np.load('samples.npy')
stats.ks_2samp(samples[:, 1], banana_samples[:, 1], alternative="two-sided", met
```

Out[ ]:
```
KstestResult(statistic=np.float64(0.00274), pvalue=np.float64(0.991915419987130
6), statistic_location=np.float64(0.3667359824411196), statistic_sign=np.int8(-
1))
```

In [ ]:
```
for i in range(2):
    print(f'KS Score p-value {i+1} : {stats.ks_2samp(banana_samples[:,i], sample
```

```
KS Score p-value 1 : 0.976033962340499
KS Score p-value 2 : 0.9916741309521137
```

As these p-values from the KS Scores are both significantly above 0.05, this indicates that the sampler is working properly.

Question 2

2.1 The derivation of SNIS estimator for the marginal likelihood

$$
\begin{aligned}
p(y_{1:T}) &= \int p(y_{1:T} \mid x)\, p(x)\, dx \\
&= \frac{\int p(y_{1:T} \mid x)\, \bar{p}(x)\, dx}{Z}, \quad \text{where } Z = \int \bar{p}(x)\, dx \\
&= \frac{\int \frac{p(y_{1:T}|x)\, \bar{p}(x)}{q(x)}\, q(x)\, dx}{\int \frac{\bar{p}(x)}{q(x)}\, q(x)\, dx} \\
&\approx \frac{\frac{1}{N}\sum_{i=1}^{N} \frac{p(y_{1:T}|x_i)\, \bar{p}(x_i)}{q(x_i)}}{\frac{1}{N}\sum_{i=1}^{N} \frac{\bar{p}(x_i)}{q(x_i)}} \\
\text{Let } W_i &= \frac{\bar{p}(x_i)}{q(x_i)},\ \omega_i = \frac{W_i}{\sum W_i}. \\
p^{\text{SNIS}}(y_{1:T}) &= \sum_{i=1}^{N} p(y_{1:T} \mid x_i)\, \omega_i, \qquad (\text{Note that } p(y_{1:T} \mid x) = \prod_{t=1}^{T} p(y_t \mid x).) \\
&= \sum_{i=1}^{N} \left(\prod_{j=1}^{T} p(y_j \mid x_i) \right) \omega_i
\end{aligned}
$$

In [ ]:
```
#This is the naive implementation
def logsumexp_naive(nums):
    return np.log(np.sum(np.exp(nums)))

#This is the stable implementation
def logsumexp_stable(values):
    max_val = np.max(values)
    return max_val + np.log(np.sum(np.exp(values - max_val)))
```

In [ ]:
```
#This is the test with large values
test_arr = np.array([-1000, -1100, -1200])

n = logsumexp_naive(test_arr)
print(' naive logsumexp:', n)
s = logsumexp_stable(test_arr)
print("stable logsumexp:", s)
```

```
 naive logsumexp: -inf
stable logsumexp: -1000.0
C:\Users\tarun\AppData\Local\Temp\ipykernel_32188\1801521852.py:3: RuntimeWarnin
g: divide by zero encountered in log
  return np.log(np.sum(np.exp(nums)))
```

We choose our test_arr to have large negative values since in our density we have terms such as $e^{-(\cdot)^2}$ which will be extremely close to zero for large values inside the quadratic term of the exponetial and thus the LSE will give $-\inf$ in the naive approach but as we see in the stable approach the LSE here gives around -1000

Log-SNIS:

For $p_1(y_j \mid x) = \mathcal{N}(y_j; x_i, \sigma^2)$, where $\sigma^2 = 0.1$:

$$
\begin{aligned}
p^{\text{SNIS}} &= \sum_{i=1}^{N} \prod_{j=1}^{T} \frac{1}{\sqrt{2\pi \cdot 0.1}} \exp\left(-\frac{(y_j - x_i)^2}{2 \cdot 0.1} \right) \cdot \omega_i \\
&= \frac{(\sqrt{5})^T}{(\sqrt{\pi})^T} \sum_{i=1}^{N} \prod_{j=1}^{T} \exp\left(-5(y_j - x_i)^2\right) \cdot \omega_i \\
&= \left(\frac{5}{\pi} \right)^{T/2} \sum_{i=1}^{N} \exp\left(-5 \sum_{j=1}^{T} (y_j - x_i)^2 + \log \omega_i \right)
\end{aligned}
$$

Log transformation for numerical stability:

$$
\begin{aligned}
\log p^{\text{SNIS}}_{\text{Gaussian}} &= \frac{T}{2} \log \frac{5}{\pi} + \text{LSE}\left(-5 \sum_{j=1}^{T} (y_j - x_i)^2 + \log \omega_i \right) \\
&= \frac{T}{2} \log \frac{5}{\pi} + \text{LSE}\left(-5 \sum_{j=1}^{T} (y_j - x_i)^2 + \log W_i \right) - \text{LSE}\left(\log W_i \right)
\end{aligned}
$$

Student's T

For $p_2(y_j \mid x) = \text{St}(y_j; x_i, \nu, \sigma^2)$:

$$
\begin{aligned}
p^{\text{SNIS}} &= \sum_{i=1}^{N} \prod_{j=1}^{T} \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right) \sqrt{\nu \pi \sigma^2}} \left(1 + \frac{(y_j - x_i)^2}{\nu \sigma^2} \right)^{-\frac{\nu+1}{2}} \cdot \omega_i \\
&= \left(\frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right) \sqrt{\frac{\nu\pi}{10}}} \right)^{T} \sum_{i=1}^{N} \prod_{j=1}^{T} \left(1 + \frac{10(y_j - x_i)^2}{\nu} \right)^{-\frac{\nu+1}{2}} \cdot \omega_i
\end{aligned}
$$

Then we take the log transformation for numerical stability:

$$
\begin{aligned}
\log p^{\text{SNIS}}_{\text{Student-t}} &= T \log \Gamma\left(\frac{\nu+1}{2} \right) - T \log \Gamma\left(\frac{\nu}{2} \right) \\
&\quad - \frac{T}{2} (\log(\pi) + \log(\nu) - \log(10)) \\
&\quad + \text{LSE}\left(\log \omega_i - \frac{\nu+1}{2} \log\left(\sum_{j=1}^{T} \left(1 + \frac{10(y_j - x_i)^2}{\nu} \right) \right) \right)
\end{aligned}
$$

Which after accounting for weight normalisation:

$$
\begin{aligned}
\log p^{\text{SNIS}}_{\text{Student-t}} &= T \log \Gamma\left(\frac{\nu+1}{2} \right) - T \log \Gamma\left(\frac{\nu}{2} \right) \\
&\quad - \frac{T}{2} (\log(\pi) + \log(\nu) - \log(10)) \\
&\quad + \text{LSE}\left(\log W_i - \frac{\nu+1}{2} \log\left(\sum_{j=1}^{T} \left(1 + \frac{10(y_j - x_i)^2}{\nu} \right) \right) \right) \\
&\quad - \text{LSE}(\log W_i)
\end{aligned}
$$

In [ ]:
```python
# Define the Banana density function
def log_banana_density(x):
    return -x[0]**2 / 10 - x[1]**2 / 10 - 2 * (x[1] - x[0]**2)**2

# Define the proposal density q(x)
def log_proposal_density(x, s):
    return - (x[0]**2)/(2*s[0]**2) - (x[1] - (x[0]**2))**2/(2*s[1]**2) - np.log(
```

Set up proceeds as follows

In [ ]:
```python
# Loading data and defining our variances
x_samples = np.load("samples.npy")
y_data = np.load("y.npy")

#Setting to 1000 runs
N = 1000

#Defining prefix variable T as in derivation
T = len(y_data)

# Generating the ssample proposals via our variances
def sample_proposal(s):
    x_1 = np.random.normal(0, s[0])
    x_2 = np.random.normal(x_1**2, s[1])
    return x_1, x_2


# Fixed parameters
sigma_sq = 0.1            # Variance of the Gaussian distribution
sigma = np.sqrt(sigma_sq)
proposal_s = [math.sqrt(5), 1/2] # Proposal distribution scaling parameters
nu = 5                    # Degrees of freedom for the Student's t-distribution

# Function to calculate the importance weight for a sample
def calc_weight(x):
    log_prior = log_banana_density(x)
    log_proposal = log_proposal_density(x, proposal_s)
    log_weight = log_prior - log_proposal
    return log_weight
```

SNIS estimator for Gaussian likelihood

In [ ]:
```python
# SNIS estimator for Gaussian likelihood
def snis_estimator_gauss(y_data, N, lse_func = logsumexp_stable):
    log_likelihoods = []  # Store log likelihood contributions for each sample
    weights = []          # Store importance weights
    x_samples = []        # Store samples from the proposal distribution

    #This will generate the N samples from the proposal distribution
    for i in range(N):
        x_samples.append(sample_proposal(proposal_s))

    #This will calculate the log-likelihood and weights for each sample
    for x in x_samples:
        log_weight = calc_weight(x)

        # The gaussian likelihood contribution to log-likelihood
        log_likelihoods.append(log_weight - 5 * np.sum((y_data - x[0])**2))
```

```
        weights.append(log_weight)

    # Performing the log-sum-exp on the log likelihoods and weights for numerica
    lse = lse_func(log_likelihoods)
    lse_w = lse_func(weights)

    # Return the normalized log marginal likelihood estimate
    return lse - lse_w - np.log(N)
```

SNIS estimator for Student's t-distribution likelihood

In [ ]:
```
# SNIS estimator for Student's t-distribution likelihood
def snis_estimator_student_t(y_data, N, v=nu, lse_func = logsumexp_stable):
    log_likelihoods = []
    weights = []
    T = len(y_data)
    x_samples = []

    # Generate N samples from the proposal distribution
    for i in range(N):
        x_samples.append(sample_proposal(proposal_s))

    # Precompute constants that are outside the log-sum-exp
    log10 = np.log(10)
    logpi = np.log(np.pi)
    lognu = np.log(v)
    log_gamma_nu_plus = gammaln((v + 1) / 2)
    log_gamma_nu = gammaln(v / 2)
    prefix = log_gamma_nu_plus - log_gamma_nu - (1 / 2) * (logpi + lognu - log10

    # Calculate log-likelihood and weights for each sample
    for x in x_samples:
        log_weight = calc_weight(x)

        # Compute the log likelihood using the Student's t-distribution formula
        log_terms = np.sum(np.log(1 + (10 * (y_data - x[0])**2) / v))
        log_likelihood = log_weight - (v + 1) / 2 * log_terms

        log_likelihoods.append(log_likelihood)
        weights.append(log_weight)

    # Perform log-sum-exp on the log likelihoods and weights for numerical stabi
    lse = lse_func(log_likelihoods)
    lse_w = lse_func(weights)

    # Return the normalised log marginal likelihood estimate
    return lse - lse_w + T * prefix - np.log(N)
```

In [ ]:
```
# Test runs
print("Gaussian under naive LSE:", snis_estimator_gauss(y_data, N, lse_func = lo
print("Student's t under naive LSE:", snis_estimator_student_t(y_data, N, nu, ls

print("Gaussian under stable LSE:", snis_estimator_gauss(y_data, N, lse_func = l
print("Student's t under stable LSE:", snis_estimator_student_t(y_data, N, nu, l
```

```
Gaussian under naive LSE: -inf
Student's t under naive LSE: -inf
Gaussian under stable LSE: -27105.62023268662
Student's t under stable LSE: -7582.409840031549
```

```
C:\Users\tarun\AppData\Local\Temp\ipykernel_32188\1801521852.py:3: RuntimeWarnin
g: divide by zero encountered in log
  return np.log(np.sum(np.exp(nums)))
```

From these test results, it is evident that relying on the naive computation is not viable for this task. The stable logsumexp function ensures that we obtain meaningful and reliable estimates for marginal likelihoods in the SNIS procedure, even when the input data or weights span several orders of magnitude.

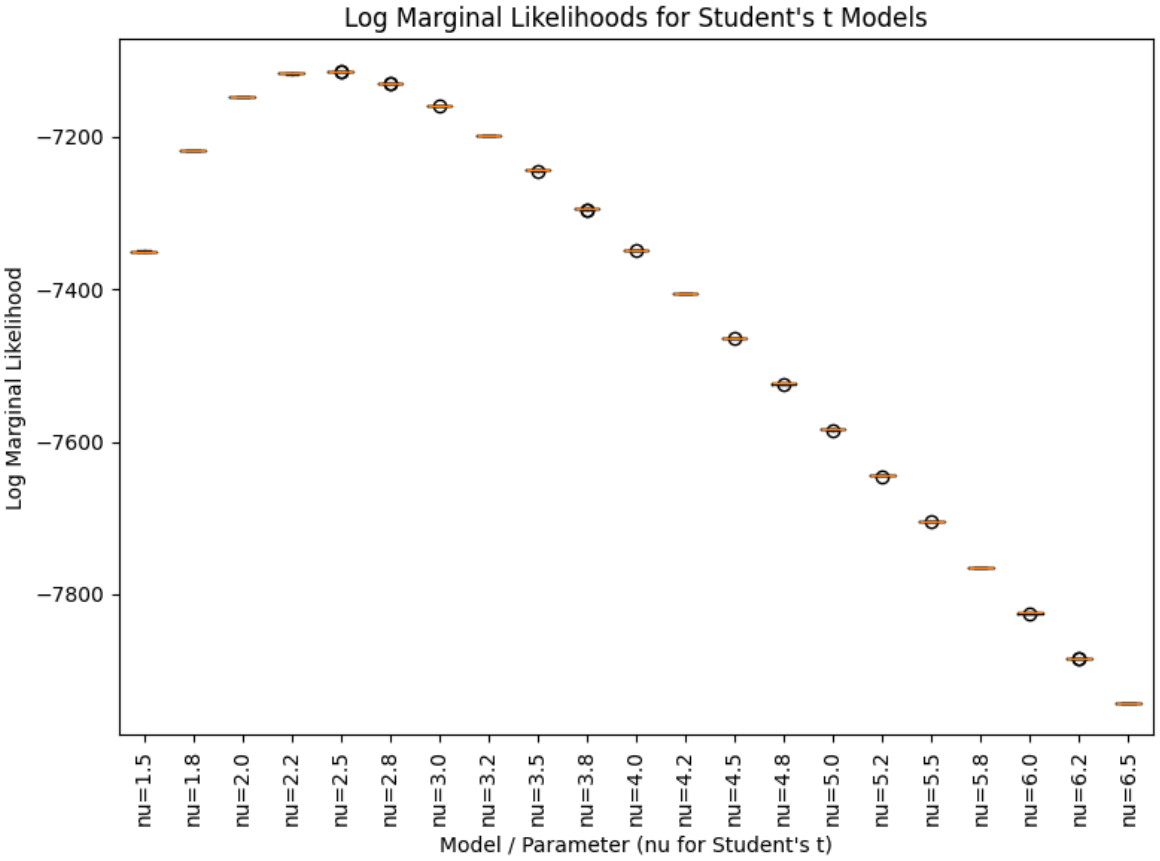
The stable implementation of the logsumexp function produced finite negative values for the log-probabilities in both marginal likelihood models. In contrast, the naive implementation returned $-\infty$ and triggered runtime warnings. This behavior highlights a critical flaw in the naive logsumexp algorithm, as it fails to handle numerical underflow effectively. This issue arises for both marginal likelihood computations, indicating a fundamental problem with the naive approach's design.

Consequently, the stable implementation of logsumexp is not just a minor optimisation—it is a necessity for this problem. It guarantees numerical stability, avoids overflow/underflow, and allows us to robustly estimate marginal likelihoods. This is particularly critical when comparing models, such as the Gaussian and Student's t-distribution likelihoods, to evaluate which better explains the observed data.

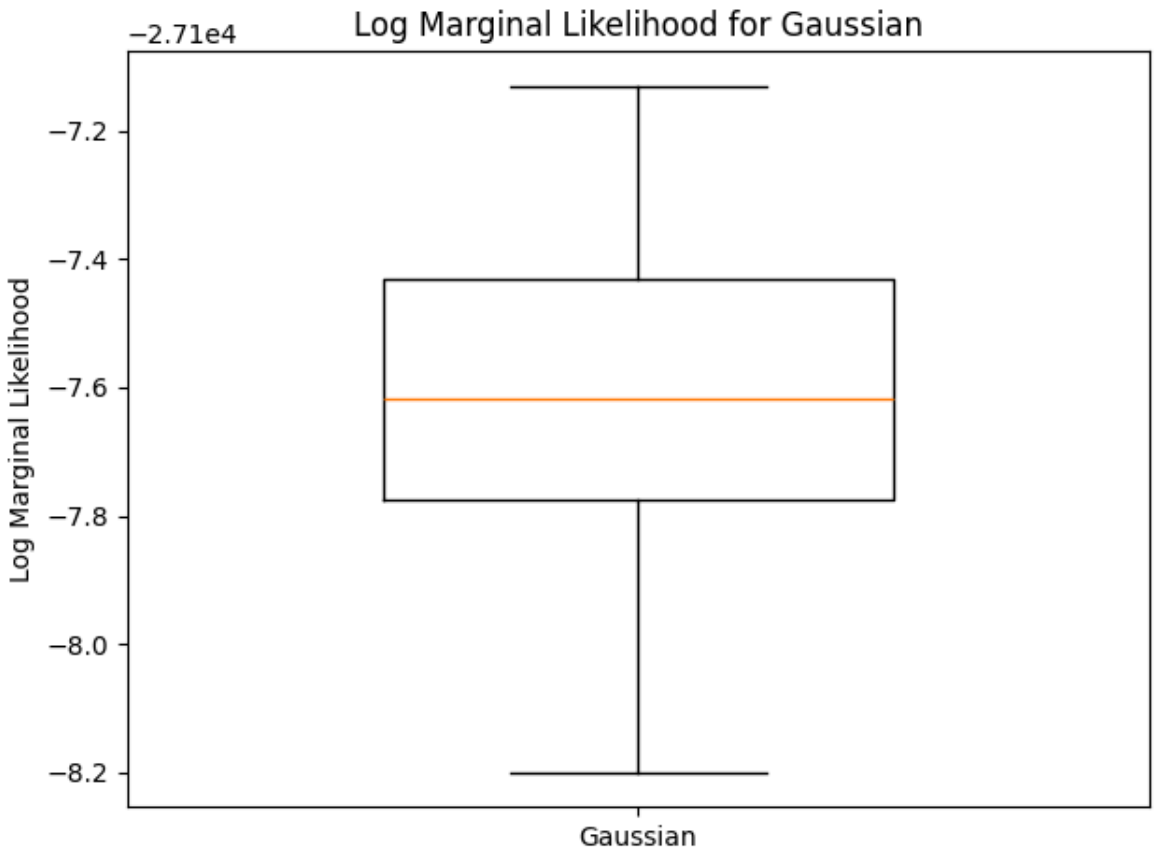
In [ ]:
```python
# Experiment parameters initialised
N = 10000
num_runs = 100
nu_values = np.linspace(1.5, 6.5, 21)  # 21 evenly spaced values of nu
sigma2 = 0.1
rng = np.random.default_rng()  # Random number generator

# Collect results
gaussian_results = []
student_t_results = {nu: [] for nu in nu_values}

# Perform 100 runs for the Gaussian model
for run in range(num_runs):
    np.random.default_rng(run)  # Set seed for reproducibility
    log_marginal = snis_estimator_gauss(y_data, N)
    gaussian_results.append(log_marginal)

# Perform 100 runs for each value of nu in the Student's t model
for nu in nu_values:
    for run in range(num_runs):
        np.random.default_rng(run)  # Set seed for reproducibility
        log_marginal = snis_estimator_student_t(y_data, N, v=nu)
        student_t_results[nu].append(log_marginal)
```

In [ ]:
```python
# Box plot visualization
plt.figure(figsize=(8, 6))

# Student's t results
positions = range(1, len(nu_values) + 1)
```

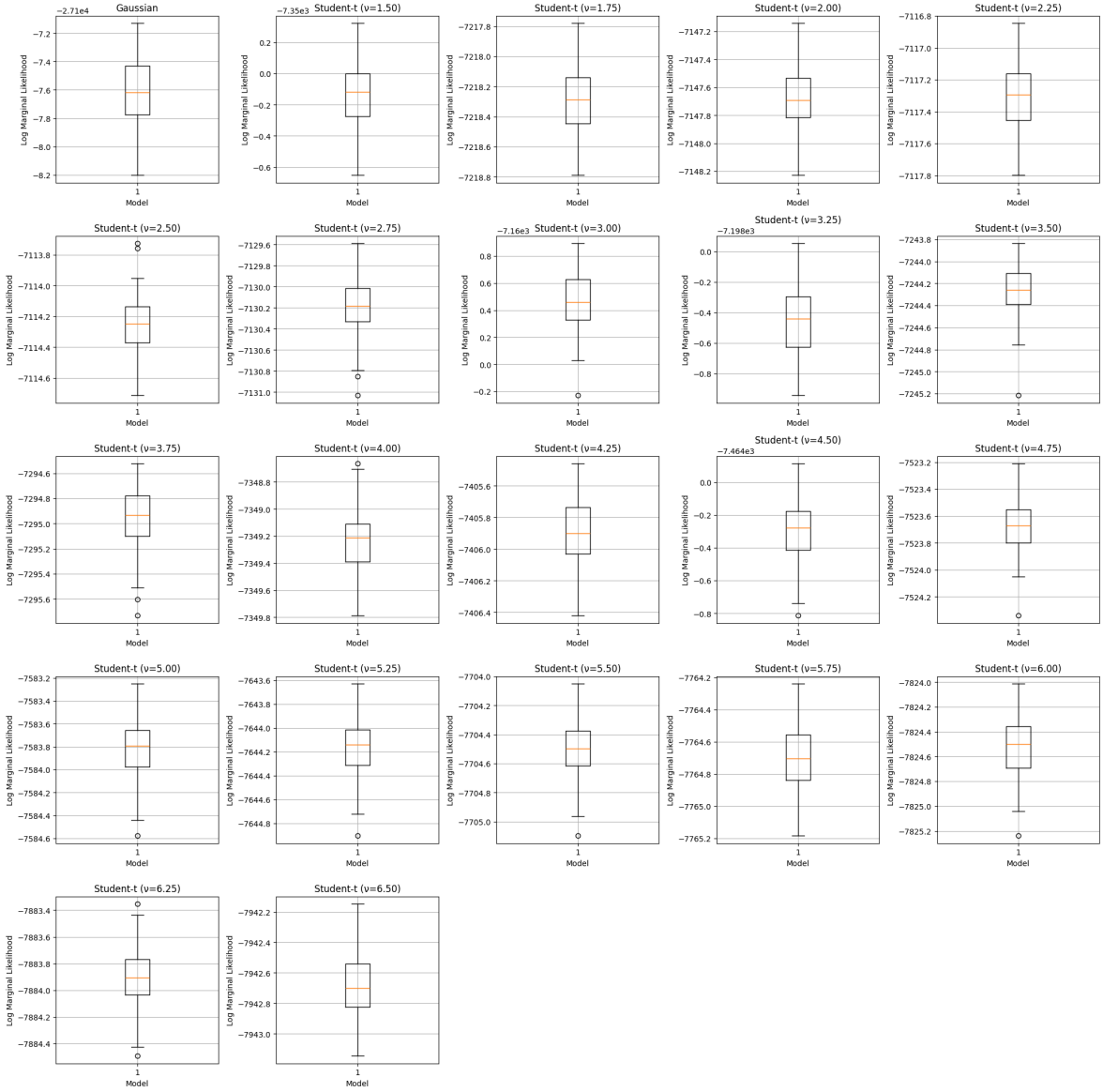
```
plt.boxplot([student_t_results[nu] for nu in nu_values], positions=positions, wi

plt.xticks(rotation=90)
plt.ylabel("Log Marginal Likelihood")
plt.xlabel("Model / Parameter (nu for Student's t)")
plt.title("Log Marginal Likelihoods for Student's t Models")
plt.tight_layout()
plt.show()
```



In [ ]:
```
plt.boxplot([gaussian_results], positions=[0], widths=0.5, tick_labels=['Gaussia

plt.ylabel("Log Marginal Likelihood")
plt.title("Log Marginal Likelihood for Gaussian")
plt.tight_layout()
plt.show()
```



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

# Create a 5x5 grid for the plots (25 total subplots, leaving 3 unused)
fig, axes = plt.subplots(5, 5, figsize=(20, 20))
axes = axes.flatten()  # Flatten the axes for easier indexing

# Plot the Gaussian box plot in the first subplot
axes[0].boxplot(gaussian_results)
axes[0].set_title('Gaussian')
axes[0].set_xlabel('Model')
axes[0].set_ylabel('Log Marginal Likelihood')
axes[0].grid(True)

# Plot the Student-t box plots in the remaining subplots
for i, nu in enumerate(sorted(nu_values)):
    ax = axes[i + 1]  # Start from axes[1]
    ax.boxplot(student_t_results[nu])
    ax.set_title(f"Student-t (v={nu:.2f})")
    ax.set_xlabel('Model')
    ax.set_ylabel('Log Marginal Likelihood')
    ax.grid(True)

# Hide any unused subplots (if less than 25 total plots are used)
for j in range(len(nu_values) + 1, len(axes)):
    fig.delaxes(axes[j])

# Adjust layout to prevent overlap
plt.tight_layout()

# Display the plots
plt.show()
```



Gaussian Model Performance:

The Gaussian model, which assumes normally distributed errors with fixed variance, is generally suitable for datasets with low variability and no significant outliers. However, the log marginal likelihood box plot for the Gaussian model shows a median value around -27600, which is significantly lower than the median values of the Student's t-distribution model at any ν. This indicates that the Gaussian model performs poorly in comparison and is not as effective in handling datasets with variability or potential outliers.

Student's t-Distribution Performance:

The Student's t-distribution model with $\nu = 2.5$ has a median log marginal likelihood around -7114, which is the highest across all models tested. This strongly suggests that the Student's t-distribution at this ν value is the most appropriate model for the data. The Student's t-distribution incorporates a degree of freedom parameter, ν, which adjusts for heavy-tailed behavior. This flexibility makes it particularly robust when dealing with datasets containing outliers or non-Gaussian noise.

- For moderate ν values, particularly in the range of $\nu \approx 2.25$ to $\nu \approx 2.75$, the log marginal likelihood values are significantly higher than those of the Gaussian model. This indicates better suitability to the data characteristics in these cases.
- At very small or very large ν values, the Student's t-distribution tends to perform similarly to the Gaussian model, as the heavy-tailed properties diminish.

Conclusion on the Best Model:

Based on the analysis, we can conclude the following:

- The **Student's t-distribution model** with an appropriately chosen ν value (e.g., $\nu \approx 2.5$) is the most effective for datasets with variability or potential outliers. This model's adaptability and robustness make it far superior to the Gaussian model in such cases.
- The Gaussian model remains a valid and computationally simpler choice for datasets that conform to its assumptions (low variability and no outliers). However, for most real-world data scenarios with irregularities, the Student's t-distribution is the better choice.

Final Recommendation:

The **Student's t-distribution with** $\nu \approx 2.5$ should be preferred, as it strikes the right balance between flexibility and robustness, making it highly effective in handling deviations from Gaussian noise assumptions.