TIF345/FYM345 Project3: A Galton board on a rocking ship

Markus Utterström Salar Ghanbari

January 27, 2025

Introduction

The Galton board, or "bean machine," is a classic demonstration of the central limit theorem, traditionally producing a binomial-like distribution of beads as they fall through a grid of pegs [1][2]. In this project, the problem is extended by introducing two additional parameters that influence the bead trajectories: an inertia parameter α, which makes it more likely for a bead to continue falling in the same direction as its previous step, and a slope parameter s, modeling a rocking base that tilts the board slightly and biases the bead distribution. Estimating these hidden parameters from observed distributions is challenging because the likelihood function $p(y_m|\alpha, s)$ is analytically intractable. Thus, we employ Approximate Bayesian Computation (ABC), a likelihood-free inference method, to infer α and s by simulating outcomes under various parameter guesses and comparing them to experimental results. To improve both the efficiency and the precision of this inference, we further integrate a neural network, trained to map observed bead distributions to parameters, thereby providing better proposals for the ABC routine. By combining simulation, Bayesian inference, and machine learning, we seek a robust and computationally tractable approach to uncovering these hidden parameters in a complex physical system.

Background, Data Generation and Simulation

A Galton board simulator was constructed to generate synthetic data. At each peg, the bead falls to the left or right with probability:

$$P_{\pm} = 0.5 \pm (\alpha M + s), \tag{1}$$

where α controls the inertial bias and s the slope-induced bias. The parameter M encodes the previous bead's direction, set to 0.5 if the bead last fell to the right and -0.5 if it fell to the left. Initially, $M = 0$ for the first peg.

We sampled α from [0, 0.5] and s from [-0.25, 0.25], ensuring a reasonable range that introduces noticeable biases without degenerating the distribution. For each (α, s) pair, multiple simulations of 1000 beads passing through 31 rows of pegs were conducted. The final positions of these beads were recorded, producing a variety of outcome distributions that reflect different underlying parameters. All simulations were performed in a Linux environment, orchestrated by Python scripts and shell commands. The output was stored as NumPy arrays for convenient retrieval and subsequent analysis, ensuring a reproducible data-generation pipeline [3].

Figure 1 presents three plots demonstrating the effect that s and also α has on the shape of the distribution. The s introduces a skewness in the distribution, a high s-value shifts the distribution to more often fall to the right. α affects how narrow the distribution is, a higher value indicates that the beads are more spread out than other ones. Using more beads would create a smoother distribution as it is now rough.

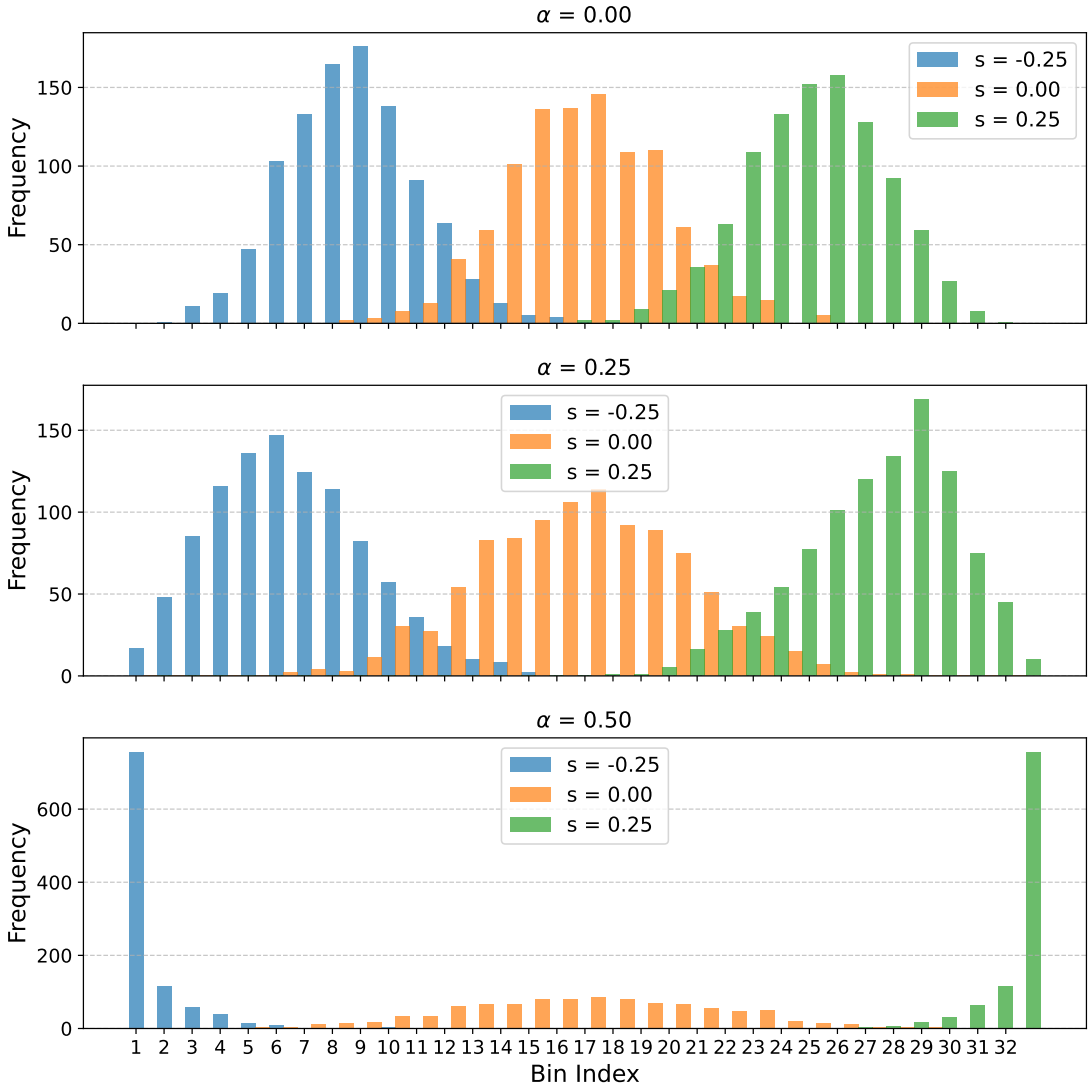

Figure 1: Simulation of 1000 beads for three different α-values and three different values of s.

Approximate Bayesian Computation (ABC) Framework

In implementing the ABC framework for this problem, the objective was to infer the most probable values of α and s by simulating synthetic data and comparing it against observed outcomes. Since directly evaluating the likelihood function $p(y_m|\alpha, s)$ is not feasible, ABC provides a likelihood-free inference method. To initialize ABC, appropriate summary statistics $s(y)$ and a kernel function $K(0)$ were selected. Based on the observation that the final bead distribution, though discrete and influenced by binomial dynamics, can be approximated by a normal distribution, the mean and variance of the normalized distribution were employed as summary statistics [4].

A Gaussian kernel was chosen to measure the distance between the observed and simulated summary statistics. The kernel bandwidth h was determined through experimentation, aiming for a balance between computational efficiency and the quality of the posterior approximation. Here, h was tuned to achieve an acceptance probability of ap-

proximately 5%. The chosen kernel has the form:

$$K\big((s(y^*) - s(y_{obs}))/h\big) = \exp\left(-\frac{|s(y^*) - s(y_{obs})|^2}{2h^2}\right). \tag{2}$$

Within the implemented code, α and s are drawn uniformly from their respective prior ranges, a Galton board simulation is run to produce a candidate dataset y^*, and $s(y^*)$ is then computed and compared with $s(y_{\text{obs}})$. The kernel value derived from this comparison serves as the acceptance probability in a rejection sampling scheme, progressively building an approximate posterior distribution for α. Concretely, α and s are sampled from uniform distributions, a simulation is generated, and its summary statistics are computed alongside those from a randomly chosen experiment y_{obs}. The Gaussian kernel, which does not require a normalization coefficient since $K(0) = 1$, determines whether the simulated result is accepted. This process repeats until a sufficient number of samples are collected to form a histogram of the posterior estimates.

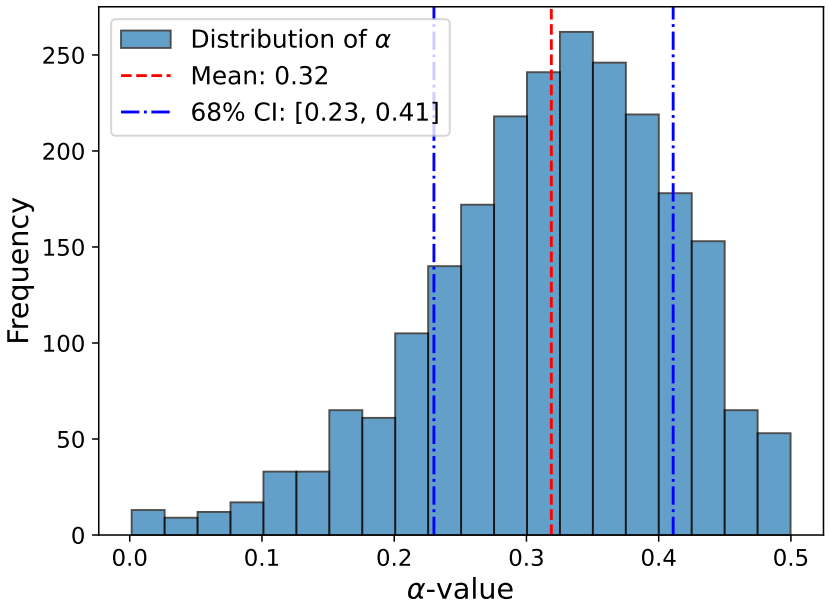

Figure 2: Histogram of sampled α values from the posterior distribution using the ABC framework.

Although this initial ABC routine successfully produces posterior estimates, it often yields wide credible intervals. This is due in part to the simultaneous inference of two parameters (α and the latent slope s) and the relatively simple summary statistics, which may not fully capture the underlying dynamics. To improve upon this approach, one can integrate a neural network to provide more informed initial guesses or use importance sampling and sequential Monte Carlo (SMC) methods to refine the posterior. However, as illustrated in Figure 2, these initial attempts without additional refinements do not strongly constrain the value of α, highlighting the need for further methodological enhancements.

Neural Network-Based Parameter Estimation

To counter the effect that the bias s has on the simulation and experimental values, a neural network is trained on 100000 simulations. The simulations was split into a training set with 80% of the data and the rest as validation data. The data was standardized as the output neurons are of different scales and it is good practice to scale data for Neural networks. The network that was used has then 32 input neurons, then two layers with first 128 and 64 neurons and finally the two output neurons. The activation function is ReLU. As α and s are on different scales, all data has been standardized for better performance of the network.

The training process is illustrated in Figure 3, which shows both training and validation root-mean-square error (RMSE) decreasing over 100 epochs, indicating stable convergence. For the validation set, the RMSE was approximately 0.0220 for α and 0.0051 for s, suggesting that the NN learned a reasonably accurate mapping from final distributions to parameters.

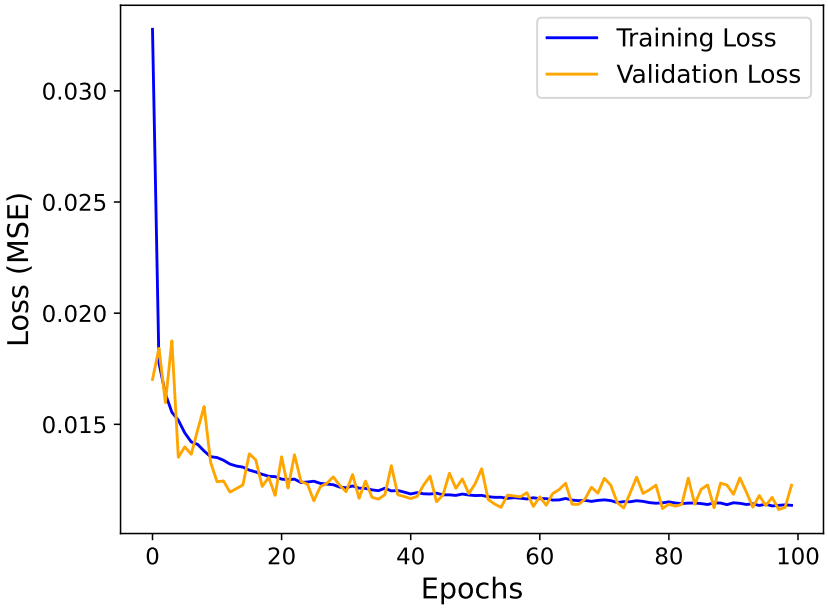

Figure 3: Training and validation loss (RMSE) over 100 epochs, showing model convergence during training.

The accuracy of the NN predictions is further demonstrated in Figure 4, where predicted versus true values are plotted for both parameters. The dashed red line represents the ideal scenario where predictions match true values exactly. While the NN performs well on both parameters, it shows notably higher accuracy for s. This is likely because changes in the slope s produce more symmetric or asymmetric shifts in the bead distribution, making s easier to identify. In contrast, the inertial bias α affects the beads' trajectories more subtly, resulting in slightly less precise estimates.

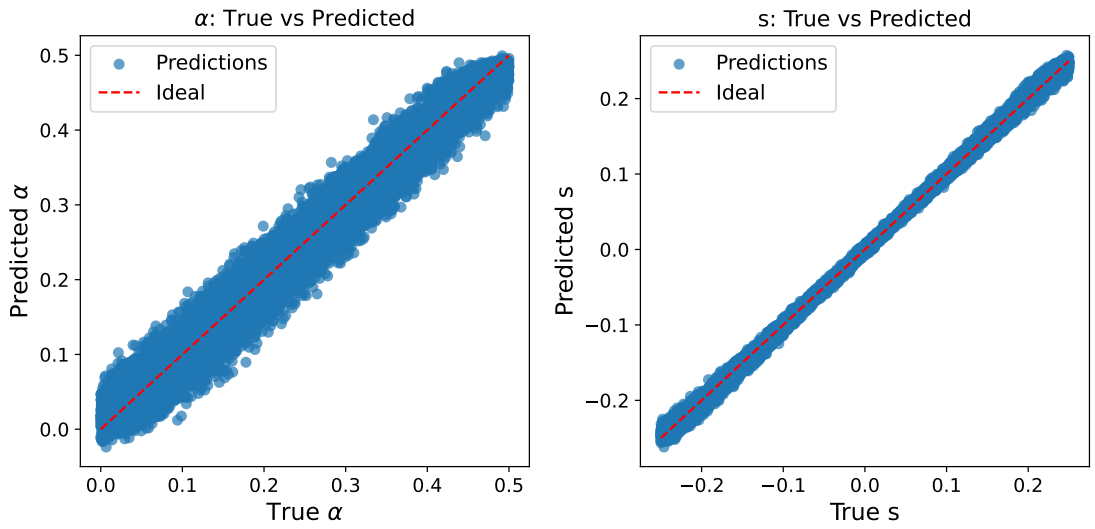

Figure 4: True vs. predicted values for α (left) and s (right), demonstrating the accuracy of the ABC model.

By providing a more accurate initial estimate for s (and a reasonable guess for α), the NN can serve as a valuable tool to guide the ABC routine. Rather than starting from a broad uniform prior, ABC can be initiated with proposals informed by the NN's predictions. This approach reduces the effective dimensionality of the search and may lead to more efficient posterior refinement, especially when combined with iterative ABC techniques.

Integrating the Neural Network into ABC

Having established that the neural network (NN) can provide accurate estimates of s and a reasonable guess for α, we now integrate these predictions into the ABC framework. The key idea is to use the NN's output to concentrate our parameter search around the predicted value for s. By replacing the broad uniform prior on s with a localized proposal distribution centered on the NN estimate, we effectively reduce the complexity introduced by the latent slope parameter.

Concretely, given an observed experiment y_{obs}, we pass it through the NN to obtain a predicted slope $\hat{s}$. To acknowledge the uncertainty in this prediction, we perturb $\hat{s}$ by drawing from an error model $g(|z - z^{\text{ML}}|)$, where z represents a candidate slope and z^{ML} could be the NN's maximum likelihood estimate for s. This ensures that we still explore a neighborhood around the NN prediction rather than treating $\hat{s}$ as fixed. The prior for s, denoted as $\pi_z(z^*)$, now reflects this localized approach, shifting from a uniform range to a distribution concentrated around $\hat{s}$.

The acceptance probability in the modified ABC step remains governed by the kernel K, but now incorporates these adjusted priors and the error model. Specifically, the acceptance probability can be written as:

$$p_{acceptance} = \frac{K((s(y^*) - s(y_{obs}))/h)\pi_z(z^*)}{g(|z - z^{\text{ML}}|)m}, \tag{3}$$

where $m = \max_{\alpha,z}(K(0)\pi_z(z^*)/g(|z - z^{\text{ML}}|))$ is a normalizing constant. Since m is independent of individual draws, it effectively cancels out as a scaling factor. This formulation

emphasizes that parameter draws closer to the NN-predicted s and consistent with the observed data y_{obs} are more likely to be accepted.

By incorporating the NN predictions, the ABC routine focuses on parameter values that yield bead distributions similar to the observed data. This informed search reduces the need to explore the entire parameter space blindly, thereby increasing efficiency and narrowing the resulting posterior distribution. As shown in Figure 5, the posterior distribution for α is notably more concentrated than in the initial ABC approach, illustrating a visibly narrower credible interval.

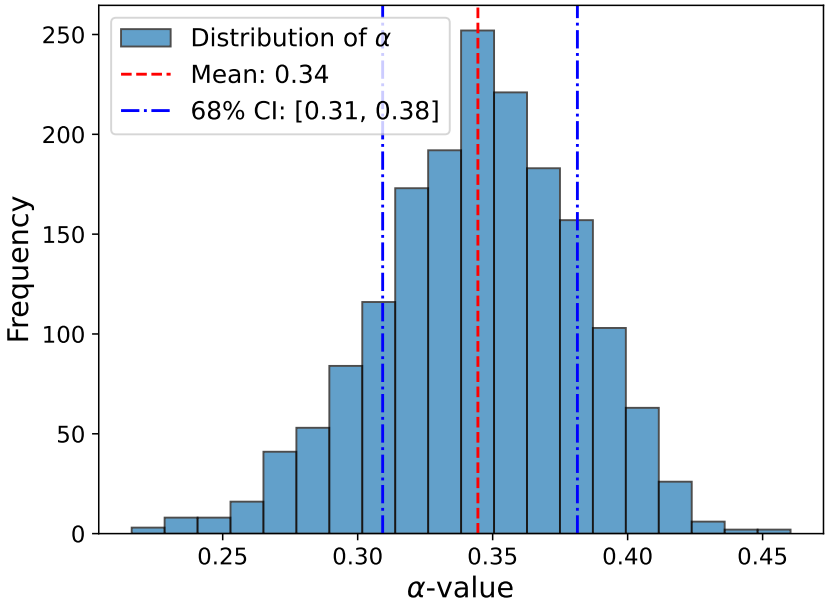

Figure 5: Posterior distribution of α after NN-guided ABC, showing a narrower 68% credible interval.

Iterative ABC

In this task, the ABC framework, combined with the neural network's predictions, is run iteratively to achieve a more refined posterior distribution for α. Instead of starting with a uniform prior, the posterior distribution from the previous iteration is sampled and used as the prior for the next. This iterative process narrows the confidence interval for α, as illustrated in Figure 6.

This iterative ABC approach effectively "zooms in" on the most likely values of α. By using the posterior from one iteration as the prior for the next, the parameter space becomes increasingly concentrated around regions of higher probability. This not only improves the acceptance rate but also significantly reduces the range of plausible parameter values that need to be explored, yielding tighter credible intervals.

Figure 6 demonstrates the results after one additional iteration of ABC. Compared to the posterior distribution shown in Figure 5, the credible interval for α is significantly narrower, reflecting the improved precision of the iterative process.

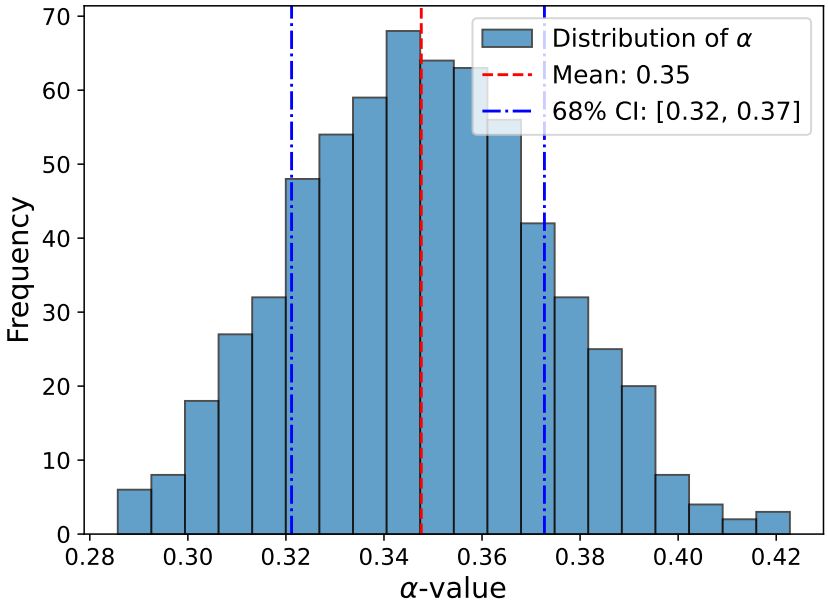

Figure 6: Posterior distribution of α after iterative ABC with a narrower credible interval.

Improvements for the ABC

There are several areas where improvements to the ABC routine could be implemented. One improvements is to increase the number of beads in the experiment, that would improve the currently used summary statistics as the central limit theorem is applied. Furthermore, different summary statistics could be tested more thoroughly to get a better understanding of the ABC method. In the basic approach that this project has been approached, this has not been explored.

The prediction for α by the NN can be implemented into the ABC with latent variable elimination. It could serve as a prior for the α values. The error associated with the NN can be included in the acceptance probability in the same manner as the s-variable.

Moreover, another methods like ABC-SMC (Sequential Monte Carlo) or ABC-MCMC (Markov Chain Monte Carlo) could be explored to adaptively refine proposals and thresholds during inference. By systematically reducing the bandwidth h, these methods focus sampling around the most plausible parameter values, accelerating convergence and improving posterior precision. These enhancements would make the ABC framework more robust, efficient, and accurate for complex parameter inference problems.

References

[1] Wikipedia contributors. The galton board. https://en.wikipedia.org/wiki/Bean_machine, 2024. Accessed: 2024-12-19.

[2] Paul Erhart, Andreas Ekström, and Arkady Gonoskov. *Advanced Simulation and Machine Learning*. Lecture notes, 2024. Chapter 10.

[3] Real Python Team. Introduction to pyenv: Manage multiple python versions and environments. https://realpython.com/intro-to-pyenv/#virtual-environments-and-pyenv, 2024. Accessed: 2024-12-19.

[4] Scott A. Sisson, Yanan Fan, and Mark Beaumont. *Handbook of Approximate Bayesian Computation*. Chapman & Hall/CRC, Boca Raton, FL, 2019.

Code

```python
# %%
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats

# Set some default plotting parameters

plt.rcParams['font.size'] = 15
plt.rcParams['axes.titlesize'] = 14
plt.rcParams['axes.labelsize'] = 15
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['legend.fontsize'] = 13

# %%
# Step 1: Simulator for the Galton Board

def galton_simulator(alpha, s, num_rows=31, num_beads=1000):
    """
    Simulates the Galton board experiment for given \u03b1 (inertia) and s (slope).

    Args:
        alpha (float): Inertia parameter.
        s (float): Slope parameter.
        num_rows (int): Number of rows in the Galton board.
        num_beads (int): Number of beads to simulate.

    Returns:
        np.ndarray: Distribution of beads in the bottom cells (32,).
    """
    positions = np.zeros(num_beads, dtype=int)
    for bead in range(num_beads):
        M = 0  # No bias at the first peg
        num_right = 0

        for _ in range(num_rows):
            prob_plus = 0.5 + (alpha * M + s)

            # Randomly choose between 0 and 1 based on the probabilities
```

```
            choice = 1 if np.random.rand() < prob_plus else 0

            if choice == 1:
                M = 0.5
                num_right += 1
            else:
                M = -0.5
        positions[bead] = num_right
    return positions

def reshape_bin_array(data):
    unique_positions, counts = np.unique(data, return_counts=True)
    final_observed_data = np.zeros(32, dtype=int)
    # Map the counts to their corresponding bins
    for pos, count in zip(unique_positions, counts):
        final_observed_data[pos] = count
    return final_observed_data


# %%
alphas = np.linspace(0, 0.5, 3)
ss = np.linspace(-0.25, 0.25, 3)

# Create subplots for all alpha values

fig, axes = plt.subplots(len(alphas), 1, figsize=(10,10), sharex=True)

bar_width = 0.5

for ax, alpha in zip(axes, alphas):
    all_observed_data = np.zeros((len(ss), 32), dtype=int)
    i = 0
    for s in ss:
        observed_data = galton_simulator(alpha, s)
        final_observed_data = reshape_bin_array(observed_data)
        all_observed_data[i] = final_observed_data
        i += 1

    # Plot overlapping bar plots for each alpha
    for idx, final_observed_data in enumerate(all_observed_data):
        x_positions = np.arange(32) + idx * bar_width  # Adjust x positions
            for overlapping
        ax.bar(x_positions, final_observed_data, width=bar_width, label=f"s = {ss[idx]:.2f}", alpha=0.7)

    ax.set_ylabel('Frequency')
    ax.set_title(rf'$\alpha$ = {alpha:.2f}')
    ax.grid(axis='y', linestyle='--', alpha=0.7)
    ax.legend()

# Add shared x-label for all subplots
plt.subplots_adjust(wspace=0, hspace=0)
plt.xlabel('Bin Index')
plt.xticks(range(32), labels=range(1, 33))  # Set x-ticks for bin indices
plt.tight_layout()
plt.savefig('example_plots.pdf')
fig.suptitle
```

```python
# %%
observed_data = galton_simulator(alpha=0.2, s=0.1)
unique_positions, counts = np.unique(observed_data, return_counts=True)

final_observed_data = np.zeros(32, dtype=int)

# Map the counts to their corresponding bins
for pos, count in zip(unique_positions, counts):
    final_observed_data[pos] = count

print(final_observed_data)

plt.figure(figsize=(12, 6))
plt.bar(range(1, 33), final_observed_data, alpha=0.8)
plt.xlabel('Feature Index')
plt.ylabel('Average Value')
plt.xticks(range(1, 33))  # Set x-ticks for each column index
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()


# %%
def summary_statistics(data,arr_bin_index):
    mean_bin_index = np.sum(data * arr_bin_index)/np.sum(data)
    variance = np.sum(data * (arr_bin_index - mean_bin_index)**2)/np.sum(data)
    #skewness = skew(data)
    return np.array([mean_bin_index, variance])

def gauss_kernel(distance, h):
    return np.exp(-0.5 / (h*h) * distance**2)

def abc_routine(experiment_data, num_samples, simulator, summary_func,
    num_rows, num_beads, h):
    """
    Approximate Bayesian Computation (ABC) routine.

    Args:
        experiment_data (np.ndarray): Observed data.
        num_samples (int): Number of prior samples.
        simulator (function): Simulator function.
        summary_func (function): Function to compute summary statistics.
        num_rows (int): Number of rows in the Galton board.
        num_beads (int): Number of beads to simulate.
        h (float): Bandwidth for the Gaussian kernel.

    Returns:
        np.ndarray: Accepted alpha parameters.
        float: Acceptance rate.
    """
    alpha_prior = np.random.uniform(0, 0.5, num_samples)
    s_prior = np.random.uniform(-0.25, 0.25, num_samples)
    accepted_params = []
    arr_bin_index = np.arange(1, 33, 1)
    accepted_count = 0

    for alpha, s in zip(alpha_prior, s_prior):
        i = np.random.randint(0, len(experiment_data))
        simulated_data = simulator(alpha, s, num_rows, num_beads)
        simulated_data = reshape_bin_array(simulated_data)

```

```
        # normalize the data
        simulated_data = simulated_data/np.sum(simulated_data)
        experiment_data[i] = experiment_data[i] / np.sum(experiment_data[i])

        simulated_stat = summary_func(simulated_data, arr_bin_index)
        observed_stat = summary_func(experiment_data[i], arr_bin_index)

        #print(simulated_stat)
        distance = np.linalg.norm(simulated_stat - observed_stat)
        kernel_prob = gauss_kernel(distance, h)

        #print(alpha, s, kernel_prob, simulated_stat, observed_stat)

        if kernel_prob > np.random.uniform(0, 1):
            accepted_params.append((alpha))
            accepted_count += 1

    return np.array(accepted_params), accepted_count/num_samples

experiment_data = np.load("board_data_.npy")

num_rows = 31
num_beads = 1000
num_samples = 40000
h = 1.8 # Bandwidth for the Gaussian kernel, about 5% acceptance rate

accepted_params, acceptance_rate = abc_routine(experiment_data, num_samples,
    galton_simulator, summary_statistics, num_rows,num_beads, h)
#print(accepted_params)
#print(acceptance_rate)
np.save('accepted_params.npy', accepted_params)



# %%
acceptance_probs = {}

h_values = [0.1, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
for h in h_values:
    accepted_params, acceptance_prob = abc_routine(experiment_data,
        num_samples, galton_simulator, summary_statistics, num_rows, num_beads,
         h)
    acceptance_probs[h] = acceptance_prob
    print(f"h = {h}, Acceptance Probability = {acceptance_prob}")


# %%
accepted_params = np.load('accepted_params.npy')

mean_alpha = np.mean(accepted_params)
ci_lower, ci_upper = np.percentile(accepted_params, [16, 84])  # 68% CI
    corresponds to 16th and 84th percentiles


print(accepted_params)
# Need to redo
plt.hist(accepted_params, bins=20, edgecolor='black', alpha=0.7, label = r'
    Distribution of $\alpha$')
plt.axvline(mean_alpha, color='red', linestyle='--', label=f'Mean: {mean_alpha
    :.2f}')
```

```
plt.axvline(ci_lower, color='blue', linestyle='-.', label=f'68% CI: [{ci_lower:.2f}, {ci_upper:.2f}]')
plt.axvline(ci_upper, color='blue', linestyle='-.')

plt.xlabel(r'$\alpha$-value')
plt.ylabel('Frequency')
plt.legend()
plt.tight_layout()
plt.savefig('abc.pdf')
# Show plot
plt.show()

print("acceptance rate: ", len(accepted_params)/num_samples)


# %%
# Create simulation data for NN
num_samples = 100000
simulated_data = np.zeros((num_samples, 32), dtype=int)
alpha = np.random.uniform(0, 0.5, num_samples)
s = np.random.uniform(-0.25, 0.25, num_samples)

for i in range(num_samples):
    print(i)
    observed_data = galton_simulator(alpha[i], s[i], 31, 1000)
    final_observed_data = reshape_bin_array(observed_data)
    simulated_data[i] = final_observed_data


np.save("simulated_data1.npy", simulated_data)
np.save("simulated_params1.npy", np.vstack((alpha, s)).T)

# %%
import tensorflow as tf
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Load the data
X = np.load("simulated_data.npy")
y = np.load("simulated_params.npy")

# Split data into training and testing sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the input features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)  # Use the same scaler for the validation set

# Optionally, standardize the target variables (if necessary)
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train)
y_val_scaled = y_scaler.transform(y_val)


# %%
# Define a simple neural network model using TensorFlow/Keras
```

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(128, activation='relu', input_dim=X_train_scaled.shape[1]),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(2)  # Output layer for two target variables (alpha and s)
])

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
history = model.fit(X_train_scaled, y_train_scaled, epochs=100, batch_size=32, validation_data=(X_val_scaled, y_val_scaled))

# Make predictions
y_pred_scaled = model.predict(X_val_scaled)

# Inverse transform predictions
y_pred = y_scaler.inverse_transform(y_pred_scaled)

# Calculate the Mean Squared Error (MSE)
mse = mean_squared_error(y_val, y_pred)
print(f'Mean Squared Error (TensorFlow): {mse}')

# %%
#model.save('regression_model.h5')
# To load it:
model = tf.keras.models.load_model('regression_model.h5')

# %%
plt.figure()
plt.plot(history.history['loss'], label='Training Loss', color='blue')
plt.plot(history.history['val_loss'], label='Validation Loss', color='orange')
#plt.title('Model Loss during Training')
plt.xlabel('Epochs')
plt.ylabel('Loss (RMSE)')
plt.legend()
plt.tight_layout()
plt.savefig('NN_training.pdf')
plt.show()

print(history.history.keys())

# %%
from sklearn.metrics import mean_absolute_error, r2_score

# Predictions on the validation set
y_pred_scaled = model.predict(X_val_scaled)

# Inverse transform predictions
y_pred = y_scaler.inverse_transform(y_pred_scaled)

# Separate true and predicted values for alpha and s
alpha_true, s_true = y_val[:, 0], y_val[:, 1]
alpha_pred, s_pred = y_pred[:, 0], y_pred[:, 1]

# Calculate errors for alpha
alpha_mse = mean_squared_error(alpha_true, alpha_pred)
alpha_mae = mean_absolute_error(alpha_true, alpha_pred)
```

```
alpha_r2 = r2_score(alpha_true, alpha_pred)

# Calculate errors for s
s_mse = mean_squared_error(s_true, s_pred)
s_mae = mean_absolute_error(s_true, s_pred)
s_r2 = r2_score(s_true, s_pred)

# RMSE
alpha_rmse = np.sqrt(alpha_mse)
s_rmse = np.sqrt(s_mse)

# Create error model for latent variable s
s_residuals = s_true - s_pred
s_residuals_mean = np.mean(s_residuals)
s_residuals_std = np.std(s_residuals)
s_residuals_gaussian = stats.norm(s_residuals_mean, s_residuals_std)

# Print error metrics
print(f"Alpha - MSE: {alpha_mse:.4f}, MAE: {alpha_mae:.4f}, RMSE: {alpha_rmse
    :.4f} R2: {alpha_r2:.4f}")
print(f"S     - MSE: {s_mse:.4f}, MAE: {s_mae:.4f}, RMSE: {s_rmse:.4f} R2: {
    s_r2:.4f}")

plt.figure(figsize=(10, 5))

# Alpha
plt.subplot(1, 2, 1)
plt.scatter(alpha_true, alpha_pred, alpha=0.7, label='Predictions')
plt.plot([min(alpha_true), max(alpha_true)], [min(alpha_true), max(alpha_true)
    ], color='red', linestyle='--', label='Ideal')
plt.title(r'$\alpha$: True vs Predicted')
plt.xlabel(r'True $\alpha$')
plt.ylabel(r'Predicted $\alpha$')
plt.legend()

# S
plt.subplot(1, 2, 2)
plt.scatter(s_true, s_pred, alpha=0.7, label='Predictions')
plt.plot([min(s_true), max(s_true)], [min(s_true), max(s_true)], color='red',
    linestyle='--', label='Ideal')
plt.title('s: True vs Predicted')
plt.xlabel('True s')
plt.ylabel('Predicted s')
plt.legend()

plt.tight_layout()
plt.savefig('NN_true_pred.pdf')
plt.show()


# %%
# ABC routine with neural network

def NN_predictor(data, model):
    # Preprocess the input distribution
    input_scaled = scaler.transform(data.reshape(1, -1))
    predicted_scaled_params = model.predict(input_scaled)
    predicted_params = y_scaler.inverse_transform(predicted_scaled_params)
    return predicted_params

```

```python

def abc_routine_nn(experiment_data, num_samples,model,error_values):
    """
    Approximate Bayesian Computation (ABC) routine with Neural network.

    Args:
        experiment_data (np.ndarray): Observed data.
        num_samples (int): Number of samples.
        model (tf.keras.Model): Neural network model.
        error_values (np.ndarray): Error values for the latent variable s.
    Returns:
        np.ndarray: Accepted alpha parameters.
    """
    alphas = np.random.uniform(0, 0.5, num_samples)
    h = 0.5
    accepted_params = []
    arr_bin_index = np.arange(1, 33, 1)

    # Error model RMSE?

    for alpha in alphas:
        i = np.random.randint(0, len(experiment_data))
        y_obs = experiment_data[i]

        # Predict the parameters using the neural network
        predicted_params = NN_predictor(y_obs, model)
        s_pred = predicted_params[0][1]

        s_pred += error_values[i]
        #print(error_model)

        simulation = galton_simulator(alpha, s_pred)
        simulation = reshape_bin_array(simulation)

        # normalize the data
        simulation = simulation/np.sum(simulation)
        y_obs= y_obs / np.sum(y_obs)

        summary_sim = summary_statistics(simulation,arr_bin_index)
        summary_exp = summary_statistics(y_obs,arr_bin_index)

        g = np.linalg.norm(summary_sim - summary_exp)
        kernel_prob = gauss_kernel(g, h)

        p_accept = kernel_prob
        print(alpha,p_accept)

        if p_accept > np.random.uniform(0, 1):
            accepted_params.append((alpha))
            #print("Accepted")

    return np.array(accepted_params)

model = tf.keras.models.load_model('regression_model.h5')
experiment_data = np.load("board_data_.npy")
error_values = s_residuals_gaussian.rvs(len(experiment_data))
num_samples = 20000
accepted_params_nn1 = abc_routine_nn(experiment_data, num_samples, model,
    error_values)

```

```
print(accepted_params_nn1/num_samples)

# %%
# Plot the histogram of accepted parameters for ABC with NN
#np.save("accepted_params_nn.npy", accepted_params_nn)

print(accepted_params_nn1)

mean_alpha = np.mean(accepted_params_nn1)
ci_lower, ci_upper = np.percentile(accepted_params_nn1, [16, 84])  # 68% CI corresponds to 16th and 84th percentiles

plt.hist(accepted_params_nn1, bins=20, edgecolor='black', alpha=0.7, label = r'Distribution␣of␣$\alpha$')
plt.axvline(mean_alpha, color='red', linestyle='--', label=f'Mean:␣{mean_alpha:.2f}')
plt.axvline(ci_lower, color='blue', linestyle='-.', label=f'68%␣CI:␣[{ci_lower:.2f},␣{ci_upper:.2f}]')
plt.axvline(ci_upper, color='blue', linestyle='-.')

plt.xlabel(r'$\alpha$-value')
plt.ylabel('Frequency')

plt.legend()
plt.tight_layout()
plt.savefig('abc_nn.pdf')
# Show plot
plt.show()

print("acceptance␣rate:␣", len(accepted_params_nn1)/num_samples)

# %%
# ABC routine with neural network


def get_prior_samples(params):
    mean = np.mean(params)
    std = np.std(params)
    return np.random.normal(mean, std, len(params))

def seq_abc_routine_nn(experiment_data,error_values,model,accepted_params_nn):
    """
    Approximate Bayesian Computation (ABC) routine with Neural network.

    Args:
        experiment_data (np.ndarray): Observed data.
        error_values (np.ndarray): Error values for the latent variable s.
        model (tf.keras.Model): Neural network model.
        accepted_params_nn (np.ndarray): Accepted parameters from the previous
            ABC run.
    Returns:
        np.ndarray: Accepted (alpha) parameters.
    """
    alphas = get_prior_samples(accepted_params_nn)
    h = 0.6
    accepted_params = []
    arr_bin_index = np.arange(1, 33, 1)

    for alpha in alphas:
        i = np.random.randint(0, len(accepted_params_nn))
```

```
        y_obs = experiment_data[i]

        # Predict the parameters using the neural network
        predicted_params = NN_predictor(y_obs, model)
        s_pred = predicted_params[0][1]

        s_pred += error_values[i]

        simulation = galton_simulator(alpha, s_pred)
        simulation = reshape_bin_array(simulation)

        simulation = simulation/np.sum(simulation)
        y_obs= y_obs / np.sum(y_obs)

        summary_sim = summary_statistics(simulation,arr_bin_index)
        summary_exp = summary_statistics(y_obs,arr_bin_index)

        g = np.linalg.norm(summary_sim - summary_exp)
        kernel_prob = gauss_kernel(g, h)

        p_accept = kernel_prob
        print(alpha,p_accept)

        if p_accept > np.random.uniform(0, 1):
            accepted_params.append((alpha))
            #print("Accepted")

    return np.array(accepted_params)

model = tf.keras.models.load_model('regression_model.h5')
experiment_data = np.load("board_data_.npy")
#num_samples = 4000
error_values = s_residuals_gaussian.rvs(len(accepted_params_nn1))
seq_accepted_params_nn = seq_abc_routine_nn(experiment_data, error_values,
    model, accepted_params_nn1)

# %%
# Plot the histogram of accepted parameters for ABC with NN
#np.save('seq_accepted_params_nn.npy', seq_accepted_params_nn)
print(seq_accepted_params_nn)

mean_alpha = np.mean(seq_accepted_params_nn)
ci_lower, ci_upper = np.percentile(seq_accepted_params_nn, [16, 84])  # 68% CI
    corresponds to 16th and 84th percentiles

plt.hist(seq_accepted_params_nn, bins=20, edgecolor='black', alpha=0.7, label
    = r'Distribution␣of␣$\alpha$')
plt.axvline(mean_alpha, color='red', linestyle='--', label=f'Mean:␣{mean_alpha
    :.2f}')
plt.axvline(ci_lower, color='blue', linestyle='-.', label=f'68%␣CI:␣[{ci_lower
    :.2f},␣{ci_upper:.2f}]')
plt.axvline(ci_upper, color='blue', linestyle='-.')
plt.xlabel(r'$\alpha$-value')
plt.ylabel('Frequency')
plt.legend()
plt.tight_layout()
plt.savefig('seq_abc_nn.pdf')
plt.show()

print("acceptance␣rate:␣", len(seq_accepted_params_nn)/num_samples)
```

```
550
551
552 # %%
553 x = np.linspace(-1, 1, 1000)
554 print(s_residuals_gaussian.pdf(x))
```