BE 521: Homework 4 Questions

Spring 2025

42 points

Due: Feb 20th, 2025

Objective: Visual responses and likelihood

AI Usage Notice

The use of artificial intelligence tools (e.g., large language models, code assistants) is permitted. However, students must explicitly state the specific ways AI was used in completing their work. Failure to disclose AI usage may result in an oral examination to assess understanding, at the discretion of Dr. Litt.

If AI was used in the completion of this assignment, please provide a statement below:

[Enter your statement here]

V1 Dataset

In this homework, you will work with data from 18 cells recorded from mouse primary visual cortex (also known as V1). Cells in this area are responsive to specific angles. Hence, a common stimulation paradigm is to show the subject a sinusoidal grating drifting at a specific angle (see included figures).

This data was collected and graciously provided by Daniel Denman in the Contreras Lab, University of Pennsylvania. The file `mouseV1.mat` contains two variables: `neurons`, a numpy array representing all the times that each of the 18 cells fired a spike during the approximately 7 minute long experiment, and `stimuli`, which provides the time (first column, in milliseconds) that a given angle (second column) was presented in this experiment. Note that each stimulus in `stimuli` is presented for exactly 2 seconds, after which a gray screen is presented for approximately 1.5 seconds (therefore each trial is approximately 3.5 seconds in duration).

In [1]:
```python
#Set up the notebook environment
from IPython.display import Image
import matplotlib.pyplot as plt
import numpy as np
from scipy.io import loadmat
import scipy.stats
```

0. Uploading Data (0 pts)

Upload the `mouseV1.mat` data to google colab. We'll load it in with the following code.

In [2]:
```python
#loadmat returns a dictionary of the data
data = loadmat('mouseV1.mat')
print(data.keys())

#get the neurons and stimuli data
neurons = data['neurons']
stimuli = data['stimuli']

#confirm that neurons contains 18 subarrays and that stimuli has two columns
print("neuron variable type: " + str(type(neurons)) + " of shape " + str(neurons.shape))
print("stimuli variable type: " + str(type(stimuli)) + " of shape " + str(stimuli.shape))
```

```
dict_keys(['__header__', '__version__', '__globals__', 'stimuli', 'neurons', 'info'])
neuron variable type: <class 'numpy.ndarray'> of shape (1, 18)
stimuli variable type: <class 'numpy.ndarray'> of shape (120, 2)
```

1. Stimulus Response (11 pts)

In this section, you will explore the response of cells to different stimulus angles.

1

How many unique rating angles, `m`, are there in `stimuli`? (1 pts)

In [3]:
```python
#your code here

unique_angles = np.unique(stimuli[:, 1])
```

```
num_angles = len(unique_angles)
# print(unique_angles)
print("number of unique angles:",num_angles)
```

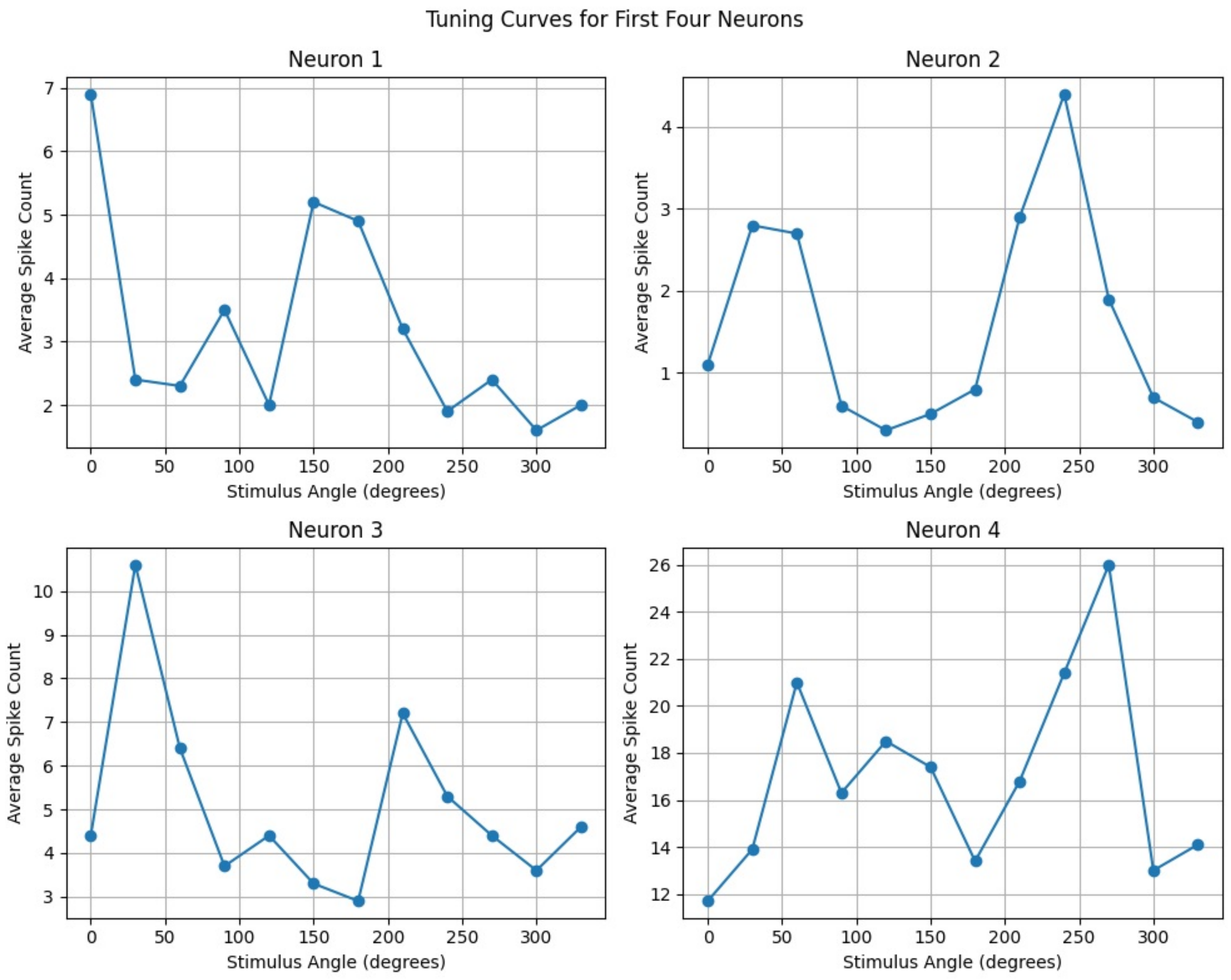
```
number of unique angles: 12
```

Your answer here

2

A *tuning curve* is frequently used to study the response of a neuron to a given range of input stimuli. To create tuning curves for this data, calculate the average number of spikes each cell fires in response to each grating angle. Store the result in an 18 × m dimensional matrix, where each element represents the response of a single neuron to a particular input stimulus angle, with each neuron assigned a row and each angle assigned a column. In a 2 × 2 Matlab subplot, plot the tuning curve for the first four cells. Place the stimulus angle on the x-axis and the number of spikes on the y-axis. (6 pts)

In [4]:
```
# Initialize a matrix to store the average spike counts
average_spikes = np.zeros((18, num_angles))

# Loop through each neuron
for neuron_idx in range(18):
    # Get spike times for the current neuron
    spike_times = neurons[0, neuron_idx]

    # Loop through each unique angle
    for angle_idx, angle in enumerate(unique_angles):
        # Find all trials where this angle was presented
        trial_times = stimuli[stimuli[:, 1] == angle, 0]

        # Count spikes during each trial
        spike_counts = []
        for start_time in trial_times:
            end_time = start_time + 2000  # Stimulus lasts 2 seconds (2000 ms)
            spikes_in_trial = np.sum((spike_times >= start_time) & (spike_times < end_time))
            spike_counts.append(spikes_in_trial)

        # Calculate the average spike count for this angle
        average_spikes[neuron_idx, angle_idx] = np.mean(spike_counts)


# Create a 2x2 subplot
fig, axes = plt.subplots(2, 2, figsize=(10, 8))
fig.suptitle('Tuning Curves for First Four Neurons')

# Plot the tuning curves
for i, ax in enumerate(axes.flat):
    if i < 4:  # Only plot the first four neurons
        ax.plot(unique_angles, average_spikes[i, :], marker='o')
        ax.set_title(f'Neuron {i+1}')
        ax.set_xlabel('Stimulus Angle (degrees)')
        ax.set_ylabel('Average Spike Count')
        ax.grid(True)

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

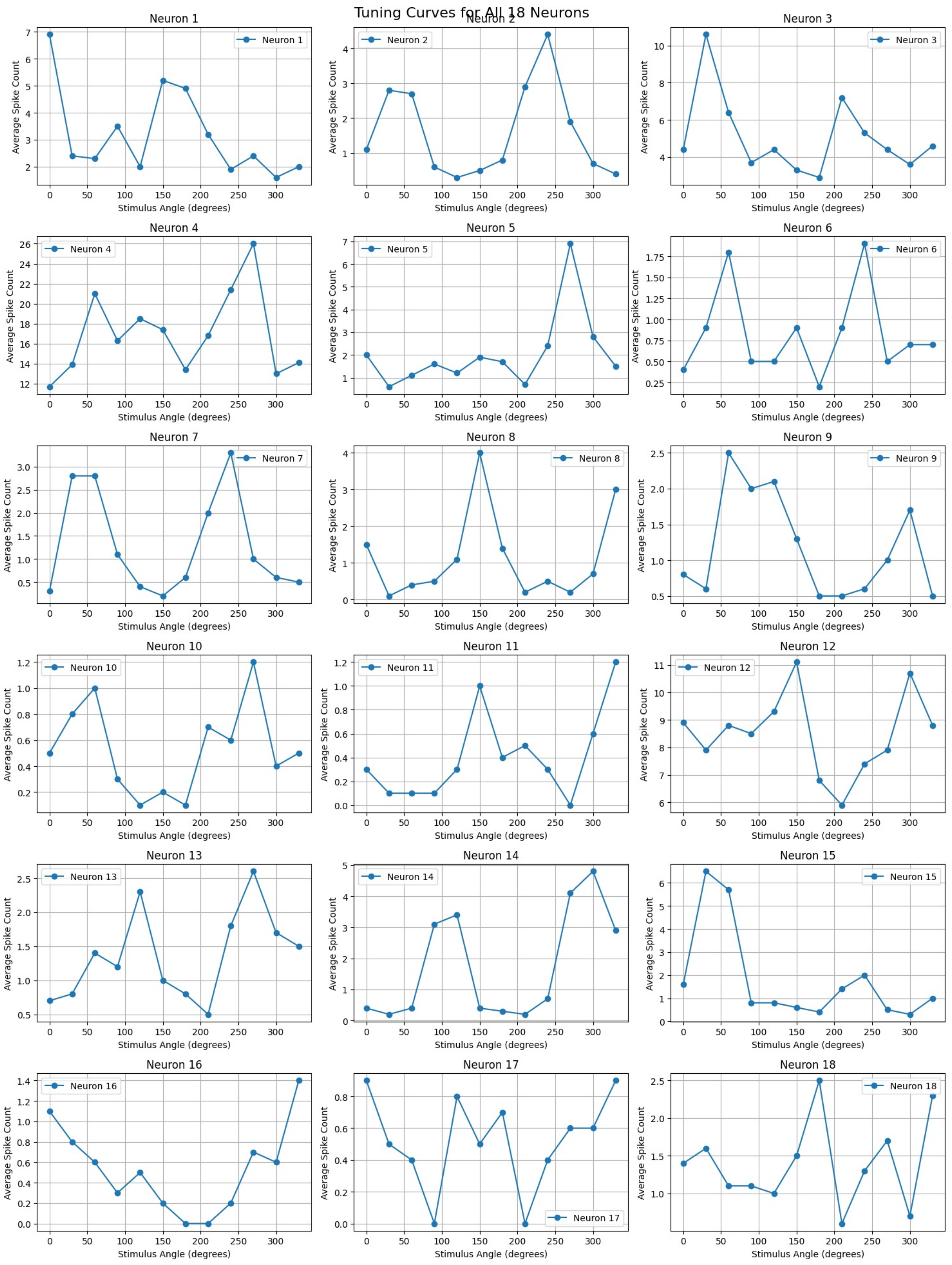


Your answer here

2a

Look through the tuning response curves of each of the 18 cells. How reasonable is it to assume that the response of a given cell to angle θ is the same as its response to angle θ + 180? Include at least a few tuning curves to back up your answer. (2 pts)

In [5]:
```python
#your code here
# Plot tuning curves for all 18 neurons
fig, axes = plt.subplots(6, 3, figsize=(15, 20))
fig.suptitle('Tuning Curves for All 18 Neurons', fontsize=16)

# Loop through each neuron and plot its tuning curve
for neuron_idx in range(18):
    row = neuron_idx // 3
    col = neuron_idx % 3
    ax = axes[row, col]
    ax.plot(unique_angles, average_spikes[neuron_idx, :], marker='o', label=f'Neuron {neuron_idx+1}')
    ax.set_xlabel('Stimulus Angle (degrees)')
    ax.set_ylabel('Average Spike Count')
    ax.set_title(f'Neuron {neuron_idx+1}')
    ax.grid(True)
    ax.legend()

plt.tight_layout()
plt.show()


# # Calculate the difference between responses at θ and θ+180°
differences = []
for neuron_idx in range(18):
    response_at_theta = average_spikes[neuron_idx, :num_angles//2]
    response_at_theta_plus_180 = average_spikes[neuron_idx, num_angles//2:]
    difference = np.mean((response_at_theta - response_at_theta_plus_180)**2)  # MSE
    differences.append(difference)

# Print the differences for each neuron
for neuron_idx, diff in enumerate(differences):
    print(f"Neuron {neuron_idx+1}: MSE between θ and θ+180° = {diff:.2f}")
```


Tuning Curves for All 18 Neurons
Neuron 1
Neuron 2
Neuron 3
Neuron 4
Neuron 5
Neuron 6
Neuron 7
Neuron 8
Neuron 9
Neuron 10
Neuron 11
Neuron 12
Neuron 13
Neuron 14
Neuron 15
Neuron 16
Neuron 17
Neuron 18
Average Spike Count
Stimulus Angle (degrees)

```
Neuron 1: MSE between θ and θ+180° = 2.74
Neuron 2: MSE between θ and θ+180° = 0.81
Neuron 3: MSE between θ and θ+180° = 2.97
Neuron 4: MSE between θ and θ+180° = 24.45
Neuron 5: MSE between θ and θ+180° = 5.43
Neuron 6: MSE between θ and θ+180° = 0.02
Neuron 7: MSE between θ and θ+180° = 0.19
Neuron 8: MSE between θ and θ+180° = 0.21
Neuron 9: MSE between θ and θ+180° = 0.92
Neuron 10: MSE between θ and θ+180° = 0.22
Neuron 11: MSE between θ and θ+180° = 0.06
Neuron 12: MSE between θ and θ+180° = 3.00
Neuron 13: MSE between θ and θ+180° = 0.47
Neuron 14: MSE between θ and θ+180° = 1.55
Neuron 15: MSE between θ and θ+180° = 6.94
Neuron 16: MSE between θ and θ+180° = 0.60
Neuron 17: MSE between θ and θ+180° = 0.14
Neuron 18: MSE between θ and θ+180° = 0.56
```

Looking at the graphs, the general trend is that the response of a given cell at theta is "almost" same as theta + 180deg as there is symmetry or similarity in the responses at θ and θ+180∘. Also by looking at the MSE error it is pretty evident as the error is very low. It is reasonable to assume based on the MSE error.

Your answer here

2b

Does this assumption have any physiological justification (given what you know about the types of cells in V1)? (2 pts)

```
Orientation Selectivity in V1:

    Neurons in the primary visual cortex (V1) are known to be orientation-selective, meaning
they respond preferentially to stimuli at specific orientations.

    However, these neurons are typically not direction-selective; they respond equally to
opposite directions of motion (e.g., a vertical grating moving left vs. right).

Symmetry in Orientation Tuning:

    Many V1 neurons exhibit symmetric tuning curves around their preferred orientation.

    This symmetry implies that the neuron’s response to an orientation θθ is similar to its
response to θ+180∘θ+180∘.

Physiological Basis:

    The lack of direction selectivity in simple V1 cells means they do not distinguish between
a grating moving in one direction versus the opposite direction.

    This property arises because simple cells in V1 respond to the spatial phase of the
stimulus (e.g., the alignment of light and dark bars) rather than the direction of motion.
```

Your answer here

2. Neural Decoding (31 pts)

Suppose we would like to work backwards - that is, for a given neural response, can we predict what the grating angle was? This process is called "neural decoding," and is especially of interest to the BCI motor control community (as we'll see in a later homework).

In this section, we will try out an approach which is detailed in Jazayeri & Movshon 2006. The method we will use involves finding the maximum likelihood of the data.

Here, the data is the number of spikes s_i that cell i fires when the subject sees a stimulus with grating angle θ. One way to think about our likelihood function is to ask the question "given a stimulus angle θ, how many spikes would I expect this cell to fire?" We can represent this number of spikes s_i using a Poisson process with parameter $f_i(\theta)$ for a stimulus θ, where f_i represents neuron i's tuning function.

A Poisson distribution is often used to model count data that occurs at a constant rate, and in this case the rate is given by $f_i(\theta)$. In other words, our likelihood function $L_i(\theta)$ for each neuron i is the probability $p(s_i | \theta)$ of neuron i firing s_i spikes for a given value of θ.

The idea in this method is to calculate the (natural) log likelihood function of each neuron and then add them all together to get the log likelihood function of the entire population of (n) neurons. We often work with the log likelihood because it allows adding of probabilities instead of multiplying, which can lead to numerical problems.

$p(s_i | \theta) \sim \mathrm{Pois}(f_i(\theta)) = f_i(\theta)^{S_i} s_i! e^{-f_i(\theta)} L_i(\theta) = p(s_i | \theta) L(\theta) = n \prod i=1 p(s_i | \theta) \log L(\theta) = n \sum i=1 \log L_i(\theta) = n \sum i=1 \log p(s_i | \theta) \propto n \sum i=1 s_i \log f_i(\theta)$

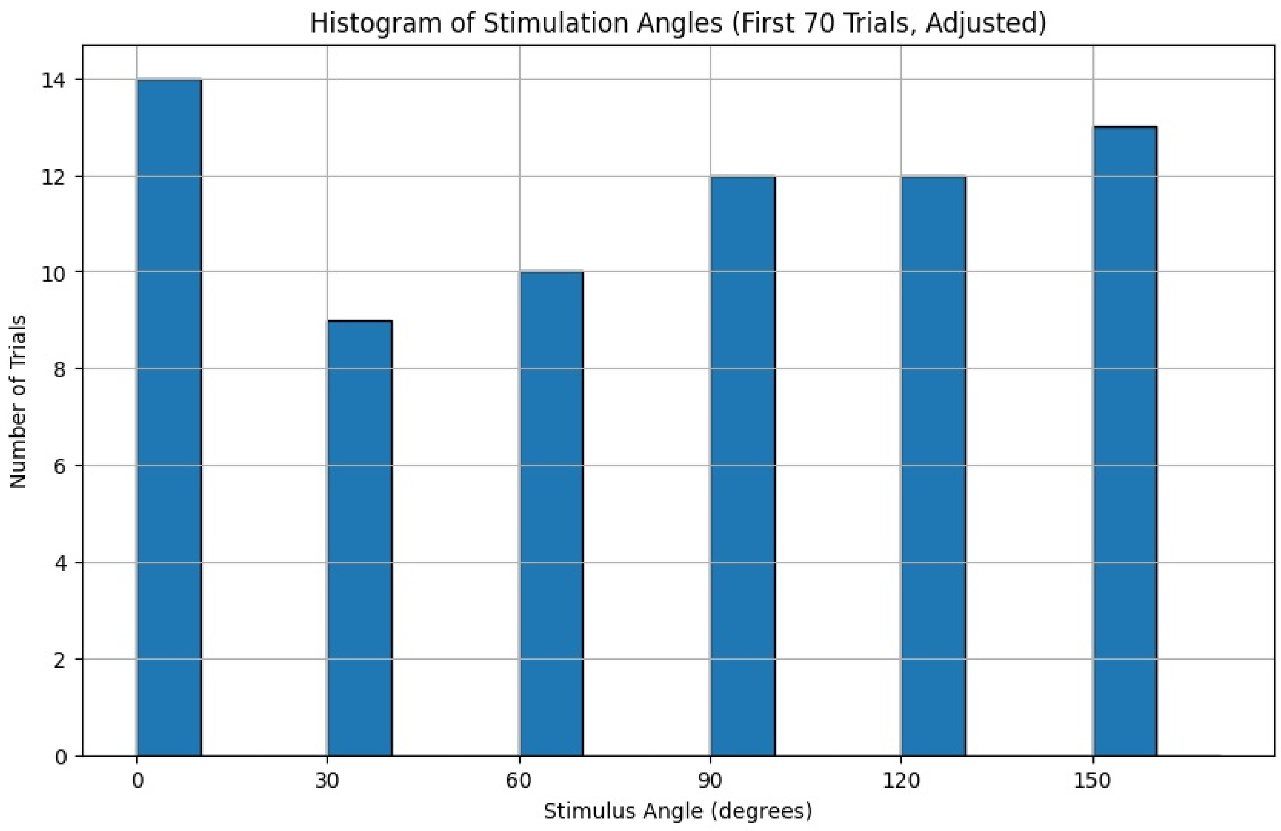
Thus, we can define the log likelihood for each neuron i as the log of its tuning curve $f_i(\theta)$ times the number of spikes s_i it fires for a particular stimulus θ, and the population log likelihood is simply the summation across all cells. This tells us that, given a set of tuning curves $f_i(\theta)$, we can compute the likelihood of observing our data s. But we already have those tuning curves for each cell from question 1.2, so all we need to know for a new (hidden) stimulus is how many spikes each neuron fires. Let s be the n-dimensional column vector of the number of spikes each cell fires after the subject is presented with a new stimulus θ' and let F be the n × m matrix representing the tuning curves of each neuron at each of the m stimuli (for us, m is the number of stimuli between 0 and 150 degrees because we assume that all neurons respond equally to θ and θ + 180 degrees). We can then compute the log likelihood of the new stimulus θ' easily using the inner product of s and F: `L = np.matmul(s.transpose(), log(F))` .

1

Compute the matrix `F` by recalculating the tuning curves you calculated in question 1.2 using only the **first 70** trials (this is akin to our "training" data).

You will use the remaining 50 trials (as "testing" data) to make predictions. Make a histogram of the number of stimulation angles for the first 70 trials to ensure that each angle (0 to 150 degrees) is presented at least a few times. (4 pts)

In [6]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Extract the first 70 trials for training
training_stimuli = stimuli[:70, :]
training_neurons = neurons # 18 x 1

# Adjust angles: for angles > 150, subtract 180
adjusted_angles = training_stimuli[:, 1].copy()  # Use .copy() to avoid modifying the original array
adjusted_angles[adjusted_angles > 150] -= 180

# Get unique stimulus angles after adjustment
unique_angles = np.unique(adjusted_angles)
num_angles = len(unique_angles)

##############################################################################################
# Recalculating F
# Initialize the tuning curve matrix F (n_neurons x num_angles)
n_neurons = 18
F = np.zeros((n_neurons, num_angles))
# Calculate the tuning curves (average spike counts) for each neuron and each angle in the training data
for neuron_idx in range(n_neurons):
    spike_times = training_neurons[0, neuron_idx]

    for angle_idx, angle in enumerate(unique_angles):
        # Find all trials where this angle was presented
        trial_times = training_stimuli[adjusted_angles == angle, 0]

        # Count spikes during each trial (use full 3.5 seconds)
        spike_counts = []
        for start_time in trial_times:
            end_time = start_time + 3500  # Stimulus lasts 3.5 seconds (3500 ms)
            spikes_in_trial = np.sum((spike_times >= start_time) & (spike_times < end_time))
            spike_counts.append(spikes_in_trial)

        # Calculate the average spike count for this angle
            F[neuron_idx, angle_idx] = np.mean(spike_counts)

# Normalize the tuning curve matrix F by dividing by its maximum value
F_normalized = F / np.max(F)

##############################################################################################


# Plot histogram of stimulation angles for the first 70 trials
plt.figure(figsize=(10, 6))
plt.hist(adjusted_angles, bins=np.arange(0, 180, 10), edgecolor='black')
plt.xlabel('Stimulus Angle (degrees)')
plt.ylabel('Number of Trials')
plt.title('Histogram of Stimulation Angles (First 70 Trials, Adjusted)')
plt.xticks(np.arange(0, 180, 30))
plt.grid(True)
plt.show()
```



Your answer here

2

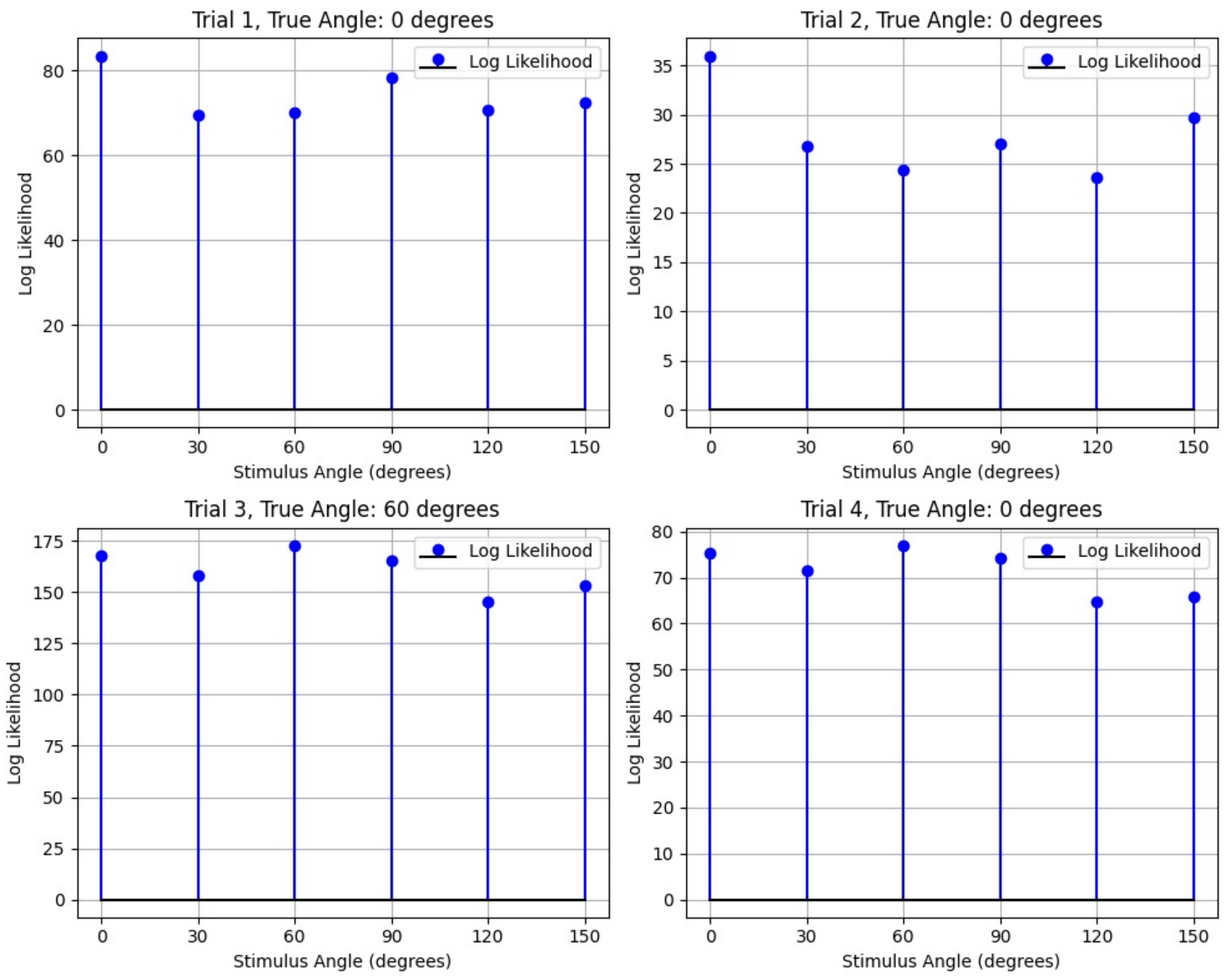
For the 50 "testing" trials, compute a n × 50 matrix **S** where each row represents the number of spikes one neuron fired in response to a each of the 50 trials.

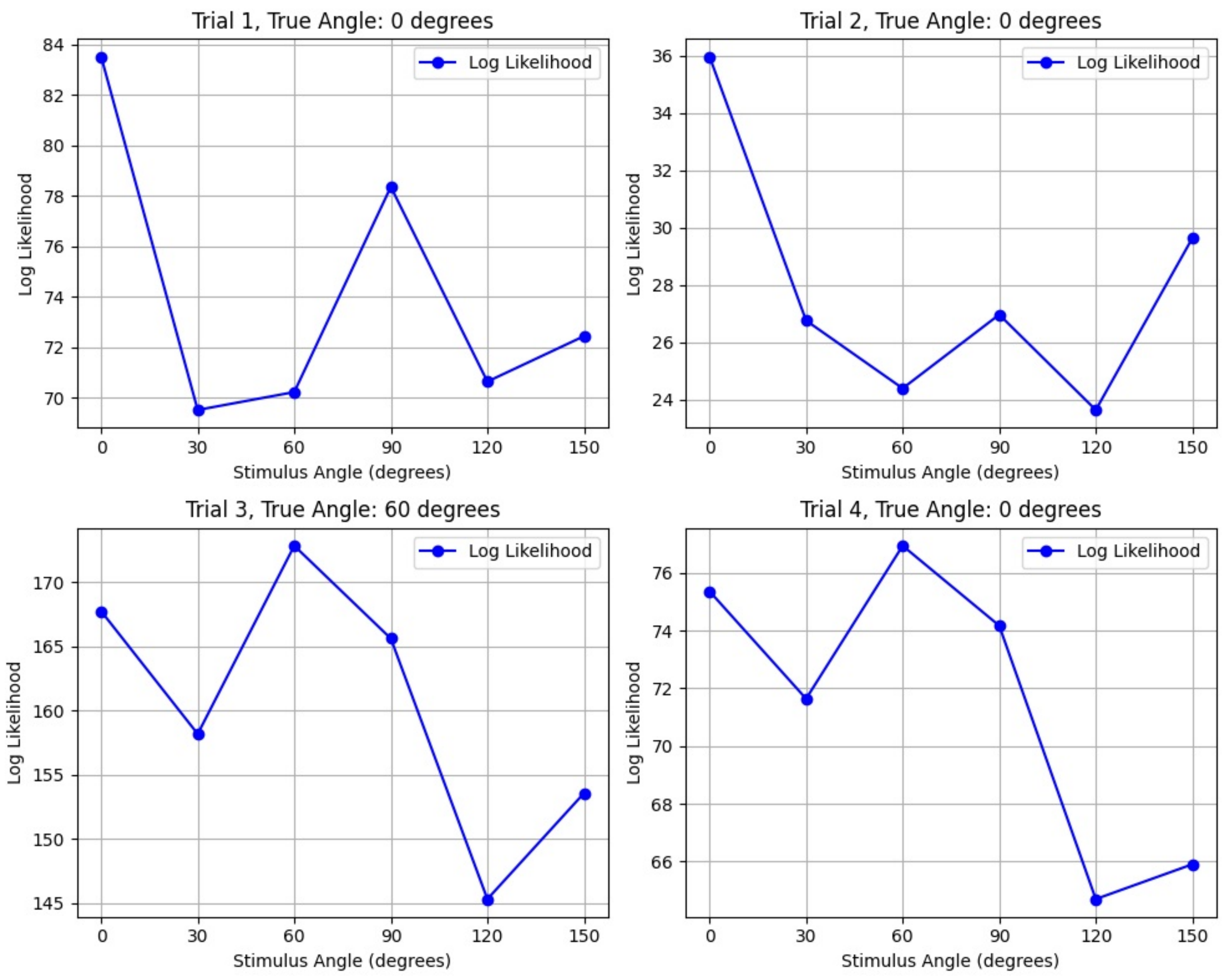
With this, you can easily compute the log likelihood functions for all the trials at once with the command: `L_test = S'*log(F)` . (Hint: add a small number to `F` to avoid taking the log of 0)

2a

Plot the likelihood functions for the first four testing trials in a 2 × 2 subplot. In the title of each plot, give the trial number (1, 2, 3, or 4) and the true stimulation angle. Make sure to label your axes correctly. (5 pts)

In [7]:
```python
# Extract the remaining 50 trials for testing
testing_stimuli = stimuli[70: 120:, :]
testing_neurons = neurons

# Small constant to avoid log(0)
epsilon = 1e-6

# Initialize a vector to store predicted angles

# Adjust angles for testing data
adjusted_angles_test = testing_stimuli[:, 1]
adjusted_angles_test[adjusted_angles_test > 150] -= 180

# Compare predicted angles to actual angles
actual_angles = adjusted_angles_test  # Use the adjusted angles for comparison


# Small constant to avoid log(0)
epsilon = 1e-6

# Initialize the spike count matrix S (n neurons x 50 testing trials)
S = np.zeros((n_neurons, testing_stimuli.shape[0]))

# Compute spike counts for each neuron in each testing trial
for trial_idx in range(testing_stimuli.shape[0]):
    spike_counts = np.zeros(n_neurons)
    for neuron_idx in range(n_neurons):
        spike_times = testing_neurons[0, neuron_idx]
        start_time = testing_stimuli[trial_idx, 0]
        end_time = start_time + 3500  # Stimulus lasts 3.5 seconds (3500 ms)
        spike_counts[neuron_idx] = np.sum((spike_times >= start_time) & (spike_times < end_time))
    # Store the spike counts in the matrix S
    S[:, trial_idx] = spike_counts

# Compute log likelihood for each trial
L_test = np.matmul(S.T, np.log(F + epsilon))  # Transpose S to match dimensions for multiplication

# Plot likelihood functions for the first four trials
plt.figure(figsize=(10, 8))

for i in range(4):
    plt.subplot(2, 2, i+1)
    # Plot the likelihoods for the current trial as a stem plot
    plt.stem(unique_angles, L_test[i, :], linefmt='b-', markerfmt='bo', basefmt='k-', label='Log Likelihood')
    plt.xlabel('Stimulus Angle (degrees)')
    plt.ylabel('Log Likelihood')
    plt.title(f'Trial {i+1}, True Angle: {testing_stimuli[i, 1]} degrees')
    plt.xticks(np.arange(0, 180, 30))
    plt.grid(True)
    plt.legend()

plt.tight_layout()
plt.show()

#####if you are expecting a line plot

# Plot likelihood functions for the first four trials
plt.figure(figsize=(10, 8))

for i in range(4):
    plt.subplot(2, 2, i+1)
    # Plot the likelihoods for the current trial as a line graph
    plt.plot(unique_angles, L_test[i, :], marker='o', linestyle='-', color='b', label='Log Likelihood')
    plt.xlabel('Stimulus Angle (degrees)')
    plt.ylabel('Log Likelihood')
    plt.title(f'Trial {i+1}, True Angle: {testing_stimuli[i, 1]} degrees')
    plt.xticks(np.arange(0, 180, 30))
    plt.grid(True)
    plt.legend()

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





Your answer here

2b

How well do these four likelihood functions seem to match the true stimulation angle? Explain in a few sentences. (3 pts)

your code here

```
Trial 1, True Angle: 0 degrees:

    The likelihood function shows a clear peak at 0 degrees, indicating that the model
correctly identifies the true stimulation angle.

Trial 2, True Angle: 0 degrees:

    The likelihood function also shows a clear peak at 0 degrees, demonstrating that the model
accurately decodes the true angle.

Trial 3, True Angle: 60 degrees:

    The likelihood function shows a clear peak at 60 degrees, which matches the true
stimulation angle.

Trial 4, True Angle: 0 degrees:

    The likelihood function does not show a clear peak at 0 degrees.
```

Your answer here

2c

Compute the maximum likelihood estimate (MLE) for each of the 50 trials. This is another way of asking which angle θ has the highest probability.

> $\hat{\theta}_{MLE}$ = argmax.

> In what percentage of the 50 trials did your MLE correctly predict the stimulation angle [0-150]? (5 pts)

In [8]:
```python
# Compute MLE for each of the 50 trials
mle_predictions = np.argmax(L_test, axis=1)  # Find the index of the max log likelihood for each trial
predicted_angles_mle = unique_angles[mle_predictions]  # Get the corresponding angles based on the indices

# Adjust the true angles as we did with the predicted angles (subtract 180 for angles > 150)
true_angles = testing_stimuli[:, 1]
true_angles[true_angles > 150] -= 180  # Adjust angles above 150 to fall within [0, 150]

# Compare the MLE predictions to the true angles
correct_predictions = predicted_angles_mle == true_angles

# Calculate the accuracy as the percentage of correct predictions
accuracy_mle = np.mean(correct_predictions) * 100

print(f"MLE Accuracy: {accuracy_mle:.2f}%")
```

```
MLE Accuracy: 42.00%
```

Your answer here

2d

In a few sentences, discuss how well this method worked for predicting the input angle from the response of the 18 neurons. What might you change in the experimental protocol to try and get a better prediction? (3 pts)

The Maximum Likelihood Estimation (MLE) method achieved an accuracy of 42.00% in predicting the input angle from the responses of the 18 neurons. While this is better than random guessing (which would have an accuracy of ~5.5% for 18 angles spaced evenly between 0 and 180 degrees), it is still relatively low. This suggests that the model is partially capturing the relationship between neural responses and stimulus angles but may be limited by several factors

```
Limited Number of Neurons:

    With only 18 neurons, the model may not have enough information to accurately decode the
stimulus angle. Increasing the number of recorded neurons could improve prediction accuracy.

Noise in Neural Responses:

    Neural responses are inherently noisy, and this noise can reduce the accuracy of the
decoding model. Averaging over more trials or using more sophisticated noise models might
help.

Tuning Curve Estimation:

    The tuning curves (FF) are estimated from a limited number of trials (70 trials).
Increasing the number of trials for training could improve the robustness of the tuning
curves.

Stimulus Presentation:

    Each stimulus is presented for only 2 seconds, which may not be enough time for the
neurons to fully encode the stimulus. Increasing the stimulus duration could improve the
signal-to-noise ratio in the neural responses.
```

To improve:

Increase the Number of Neurons:

```
Record from more neurons in V1 to provide the model with more information about the stimulus.
```

Increase the Number of Trials:

```
Collect more trials for each angle to improve the estimation of tuning curves and reduce the
impact of noise.
```

Increase Stimulus Duration:

```
Present each stimulus for a longer duration (e.g., 3–4 seconds) to allow the neurons to fully
encode the stimulus.
```

Use More Advanced Decoding Methods:

```
Consider using more sophisticated decoding methods, such as Bayesian decoding or machine
learning models (e.g., neural networks), which may better capture the relationship between
```

```
neural responses and stimulus angles.
```

Your answer here

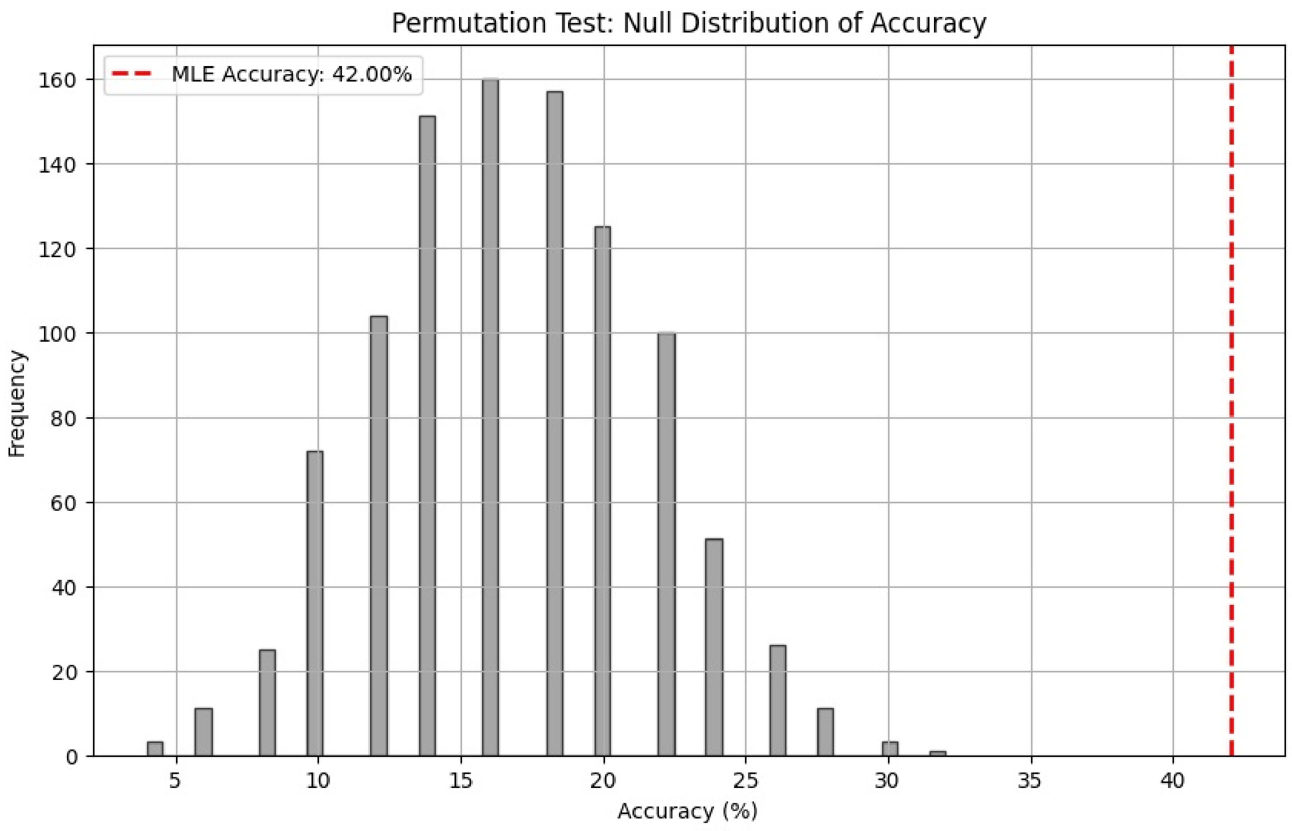
3

It is important to show that your findings are not a result of chance. One way to demonstrate this is using a "permutation test." Here, we will perform a permutation test by randomly reassigning new grating angles to the 50 test responses and then calculating how often the new grating angles match the true stimulation angles.

3a

Simulate the chance prediction (the "null" distribution) by randomly reassigning the stimulation angles 1000 times. For each permutation, calculate the percent accuracy of each label. Create a histogram of the 1000 accuracy measurements for the null distribution. Add a red vertical line with the accuracy calculated from using the Jazayeri method above. Hint: You should see a very recognizable type of distribution. If you're having trouble seeing it, try increasing the number of histogram bins. (4 pts)

In [9]:
```python
# Number of permutations
num_permutations = 1000

# Initialize list to store accuracy for each permutation
permutation_accuracies = []

# Perform the permutation test
for _ in range(num_permutations):
    # Shuffle the true angles to simulate random chance
    shuffled_angles = np.random.permutation(true_angles)

    # Compare shuffled angles to the predicted angles
    correct_predictions_permutation = predicted_angles_mle == shuffled_angles

    # Calculate the accuracy for this permutation
    accuracy_permutation = np.mean(correct_predictions_permutation) * 100
    permutation_accuracies.append(accuracy_permutation)

# Plot the histogram of permutation accuracies
plt.figure(figsize=(10, 6))
plt.hist(permutation_accuracies, bins=50, edgecolor='black', color='gray', alpha=0.7)
plt.axvline(accuracy_mle, color='red', linestyle='dashed', linewidth=2, label=f'MLE Accuracy: {accuracy_mle:.2f}
plt.xlabel('Accuracy (%)')
plt.ylabel('Frequency')
plt.title('Permutation Test: Null Distribution of Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```



Your answer here

3b

Is the null distribution what you expected? Explain. (1 pt)

Yes, the null distribution is what I expected because it represents the accuracy that would be achieved by random guessing. For 18 angles, random guessing would yield an accuracy of approximately 15.5%, and the null distribution is centered around this value. The MLE accuracy of 42.00% is significantly higher than the null distribution, indicating that the model is capturing a meaningful relationship between the neural responses and the stimulus angles. This confirms that the MLE method is not simply performing at chance level but is instead decoding the stimulus angles based on the neural data.

Your answer here

3c

What is the probability that your actual accuracy measurement comes from the null-distribution? That is, calculate the fraction of permutation samples with accuracy *more extreme* relative to the mean of the null distribution (less than your measurement if your value is less than the mean, or more than your measurement if your value is more than the mean). You may find functions in the already-imported `scipy.stats` package to be helpful. (2 pts)

```
In [10]: import numpy as np
         from scipy.stats import percentileofscore

         # Example null distribution (replace with your actual null distribution)
         null_distribution = permutation_accuracies

         # Observed accuracy
         observed_accuracy = 42.00

         # Calculate the p-value
         p_value = 1 - percentileofscore(null_distribution, observed_accuracy) / 100

         print(f"p-value: {p_value:.4f}")
```

```
p-value: 0.0000
```

Your answer here

3d

What would the probability be if your accuracy had been 25%? (1 pt)

```
In [11]: #your code here
         # Observed accuracy
         observed_accuracy = 25.00

         # Calculate the p-value
         p_value = 1 - percentileofscore(null_distribution, observed_accuracy) / 100

         print(f"p-value: {p_value:.4f}")
```

```
p-value: 0.0410
```

Your answer here

3e

What is this value typically called? (1 pt)

```
The value calculated in this context is called the p-value. It represents the probability of
obtaining an accuracy as extreme as or more extreme than the observed accuracy (25.00%) under
the null hypothesis (i.e., assuming no relationship between neural responses and stimulus
angles). A small p-value (e.g., p<0.05p<0.05) indicates that the observed accuracy is unlikely
to have occurred by random chance, providing evidence against the null hypothesis.
```

Your answer here

4

The tuning curves and neuron responses to a new trial were calculated using the number of spikes each neuron fired after the stimulation. But what if a particular neuron just happens to fire a lot and fires even more when it gets a stimulus to which it's tuned? Those neurons could "swamp" the log likelihood estimate given in Equation 1 by virtue of just having a larger average s_i and f_i(\theta). How might we correct for this type of behavior? Suggest a possible method. (2 pts)

Your answer here

To correct for neurons with high firing rates dominating the log-likelihood estimate, we can normalize the spike counts or standardize the tuning curves. One effective method is to convert the spike counts (si) to z-scores for each neuron by subtracting the mean and dividing by the standard deviation of spike counts across all trials. This ensures that all neurons contribute equally to the log-likelihood calculation, regardless of their baseline firing rates or variability. Mathematically, this is expressed as:

siz-score= (si−μi) / σi

where μi is the mean spike count for neuron ii across all trials, and σi is the standard deviation of spike counts for neuron ii across all trials. This approach prevents high-firing neurons from "swamping" the log-likelihood estimate and ensures a more balanced contribution from all neurons.