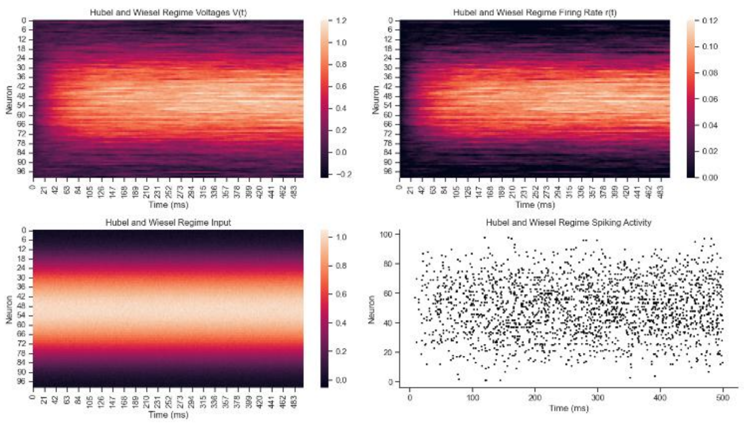
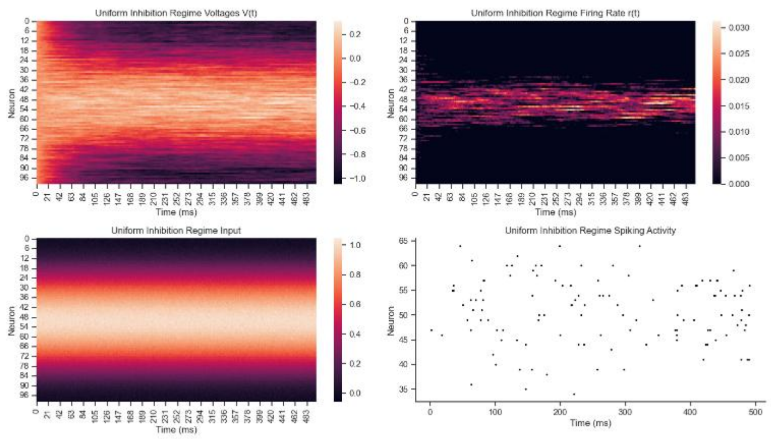
Feedforward and Recurrent Mechanisms for Orientation Selectivity in Visual Cortex B285374

1. Visualisation of Network Responses in Different Regimes

1.1 Single Trials

This section contains plots from single simulation trials for three different regimes, showing differences between them and similarities to their inputs (input = u(t) + sqrt(τ/δ)tσq(t)). Connections to recurrent connectivity and threshold nonlinearity are also discussed.





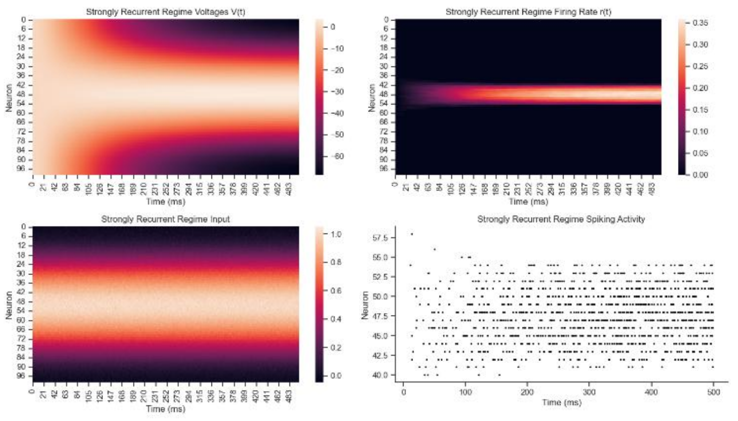


Fig. 1.1.1(upper left): Heatmaps of voltages, firing rates, and inputs, along with the spiking activity for the Hubel and Wiesel regime (feedforward input only, W0=0, W1=0)

Fig. 1.1.2(upper-right): Heatmaps of voltages, firing rates, and inputs, along with the spiking activity for the Uniform Inhibition regime (feedforward input only, W0=-4, W1=0)

Fig. 1.1.3(lower-left): Heatmaps of voltages, firing rates, and inputs, along with the spiking activity for the Hubel and Wiesel regime (feedforward input only, W0=-10, W1=11)

Fig. 1.1.1 shows a Hubel-Wiesel regime where there is a gradual, continuous voltage increase across most neurons. The firing rates are weakly tuned, with spiking distributed across all neurons. The responses almost directly reflect the input with minimal changes, showing the nature of a feedforward regime with no connection weight.

Fig. 1.1.2 shows a Uniform Inhibition regime where neurons inhibit each other equally, with negative voltages dominating away from the preferred orientation. Firing rates are low with minimal, infrequent spiking; only neurons near the preferred orientation (s=0, around neuron 50) spike occasionally, where feedforward input is strongest. Compared to the input, the Uniform

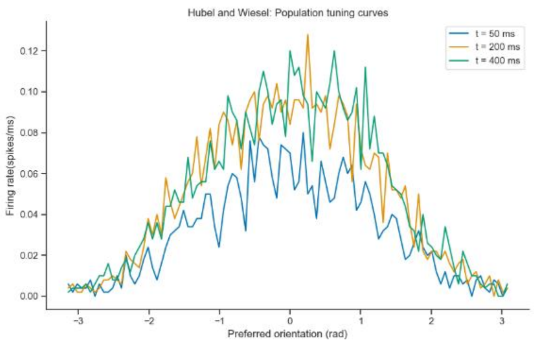
Inhibition regime eliminates most activity and only allows the most strongly driven neurons to be active (where the preferred orientation is).

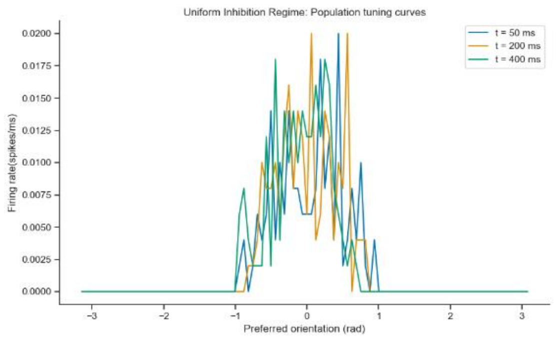
Fig. 1.1.3 shows a Strongly Recurrent regime with a delayed onset on voltages followed by the rapid centering of neurons around s = 0(through amplifying neurons near the stimulus (W1>0) along with inhibition of others (W2<0)). Firing rates show sharp, focused activity after ~50ms, with a tight clustering of spikes around s = 0 (organised activity). The heatmaps show a more focussed version of the input around the preferred orientation.

The role of recurrent connectivity (where neurons influence each other) can be seen in the Uniform Inhibition and Strongly Recurrent regimes. In the former, recurrent connectivity suppresses overall activity and narrows the response, whereas in the latter, it amplifies and sharpens responses at and around the preferred stimulus, through the selective excitation of similarly tuned neurons. Additionally, threshold nonlinearity ensures there is no firing/spiking activity when the voltage < 0 and is critical for sparse coding in the Uniform Inhibition regime. Additionally, threshold nonlinearity allows for the Strong Recurrent regime to maintain focused.

1.2 Tuning Curves

This section analyses tuning curves (where $f(s = 0, t) = \langle n(t)/\delta t \rangle$) for all three regimes, created from 50 trials with different seeds for the noise term. It also discusses how tuning curves evolve over time (50, 200, and 400ms) in each regime, and why the tuning curves for s=0 is enough to determine the single neuron tuning curves $f_i(s)$ without simulating responses to each stimulus.





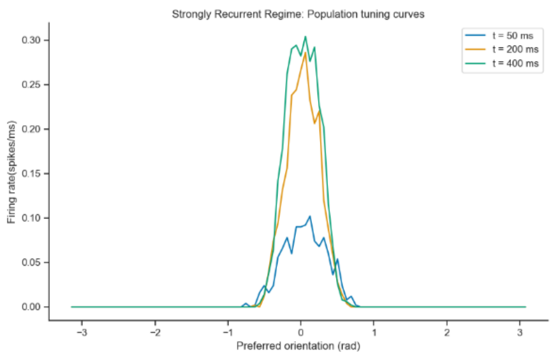


Fig. 1.2.1(upper left): Tuning curves of the Hubel-Wiesel regime at t=50ms, t=200ms, t=400ms

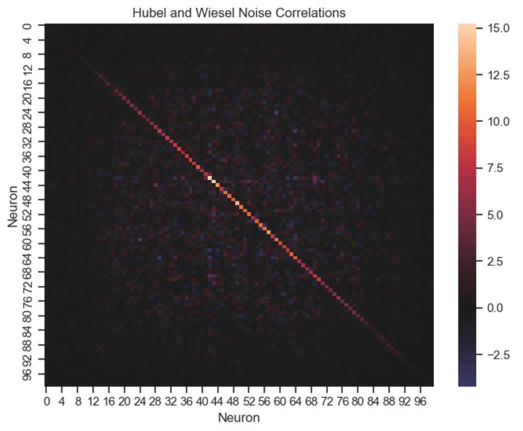
Fig 1.2.2(upper right): Tuning curves of the Uniform Inhibition Regime at t=50ms, t=200ms, t=400ms

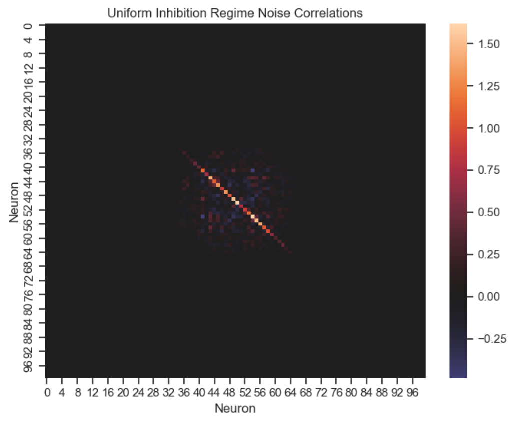
Fig 1.2.3(bottom left): Tuning curves of the Strongly Recurrent Regime at t=50s, t=200ms, t=400ms

During the Hubel-Wiesel regime (Fig. 1.2.1), the tuning curves broaden and increase in amplitude as activity accumulates. The shape remains mostly constant, just scaled up. For the Uniform Inhibition regime (Fig 1.2.2), it is very noisy with low, sparse firing rates, with some narrowing of the curve as preferred neurons cross the threshold. The strongly recurrent regime (1.2.3) starts with a lower firing rate but becomes sharpened with a higher narrow peak at s=0, and much lower noise. The Ring Network Model has circular symmetry in connectivity (W_{ij} only depends on s_i-s_j : $W_{ij} = W_0 + W_1 \cos(s_i - s_j)$) and input (ui depends on s-si: $u_i(s) = u_0+u_1 \cos(s-s_i)$). For stimulus s=0, the population response pattern across preferred orientations is identical to any single neuron's tuning curve across stimuli ($f_i(s=0) = f(0- s_i)$ due to translational symmetry in the ring network).

1.3 Noise Correlations

This section contains noise correlation ($\Sigma_{ij} = \langle(\bar{n}_i - \langle \bar{n}_i\rangle)(\bar{n}_j - \langle \bar{n}_j\rangle)\rangle$) heatmaps for each regime, discussing them and how the form of the noise correlations reflects the network dynamics and response properties. This section also explores how the noise correlations can influence neural coding for stimulus orientation in each network.





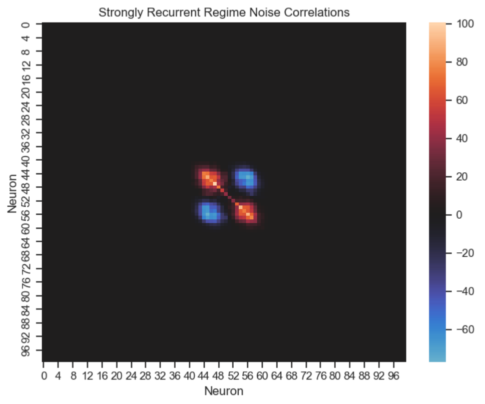


Fig. 1.3.1 (upper left): Noise correlation heatmap for the Hubel-Weibel regime

Fig. 1.3.2 (upper right): Noise correlation heatmap for the Uniform Inhibition regime

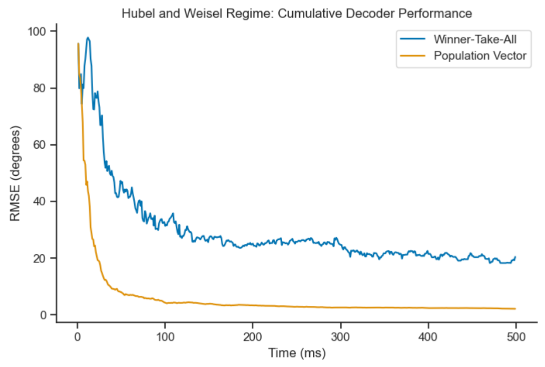
Fig. 1.3.3 (lower left): Noise correlation heatmap for the Strongly Recurrent regime

The regimes show drastically different covariance structures and magnitudes. In the Hubel-Wiesel regime (Fig. 1.3.1), there seems to be weak, unstructured correlations throughout. This reflects independent feedforward noise with broad activity across neurons. For the Uniform Inhibition regime (Fig 1.3.2), covariance values seem to be small and sparse, likely because of low spike activity in the measurement window. In the Strongly Recurrent regime (Fig 1.3.3), there are strong correlations near the centre (neuron 50) because of amplified neurons close to the stimulus. These noise correlations can significantly affect neural coding efficiency. When similarly tuned neurons have correlated noise, stimulus information may become more redundant, reducing the effectiveness of the network. This is evident in the Strongly Recurrent regime; despite sharper tuning curves, coding efficiency may be reduced with increased correlation for the stimulus identity. The other two regimes, however, have weak, unstructured correlations, which is optimal for coding efficiency. This leads to a more favourable correlation structure but limits total information in Uniform Inhibition's case, due to extreme sparsity.

2. Performance of Decoders

2.1 Population Response

This section discusses the performance of the winner-take-all and the population vector decoders from the stimulus onset up to each time *t*. The performance is measured by measuring the root mean squared error (RMSE) as a function of *t*, using cumulative spike counts $(\sum_{t' \leq t} n_i(t'))$.



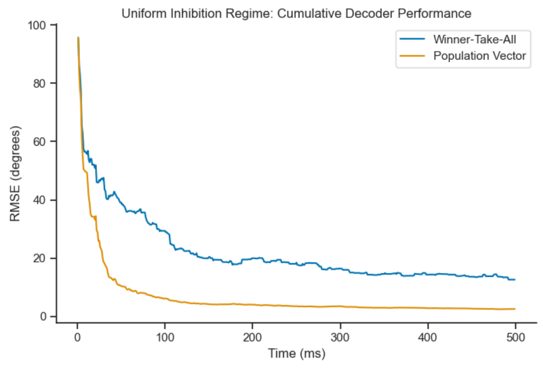


*Fig 2.1.1(upper left): **Performance of the winner-take all and population vector decoders for the Hubel-Wiesel regime, using cumulative spike counts.***

*Fig 2.1.2(upper right): **Performance of the winner-take all and population vector decoders for the Uniform Inhibition regime, using cumulative spike counts.***

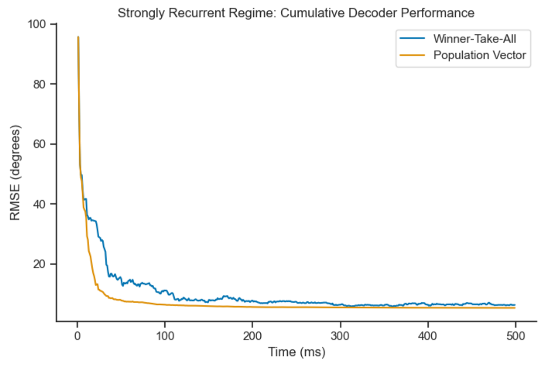
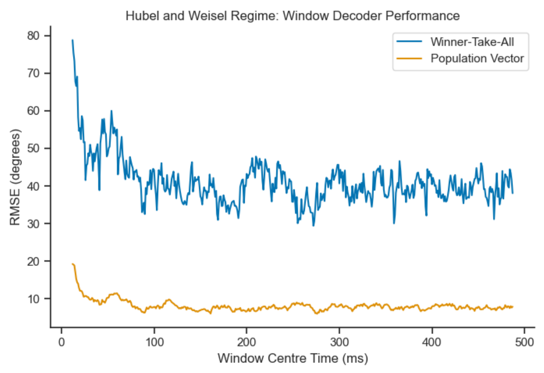


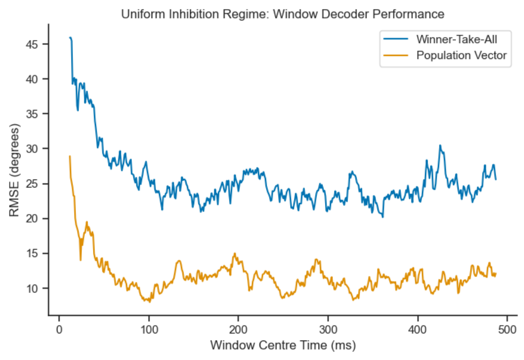
*Fig 2.1.3(lower left): **Performance of the winner-take all and population vector decoders for the Strongly Recurrent regime, using cumulative spike counts**.*

Population vector decoders prove to be superior to the winner-take-all method for all regimes, having a lower RMSE. Winner-take-all struggles the most with only feedforward input and low tuning (Fig. 2.1.1) but performs somewhat better in the Strongly Recurrent regime (Fig. 2.1.3). Population Vector decoders tend to have high RMSE values initially but drop drastically over time. Cumulative decoding benefits from integration as it effectively implements averaging over time to reduce noise. Overall, the Strongly Recurrent regime performs the best for both decoders using cumulative spike counts, providing the cleanest signal for decoding.

2.2 Optimal Decoders

This section repeats the decoding analysis of 3.1 but utilises the two decoders within time windows $t' \in [t_0, t_0 + \Delta t]$ of width $\Delta t = 25$ms and with centres $t_c = t_0 + \Delta t/2$ covering the full 500ms response period. To compare the winner-take-all and population vector decoders, the RMSE of both is computed. Additionally, this section comments on how its results compare to those obtained in 3.1.





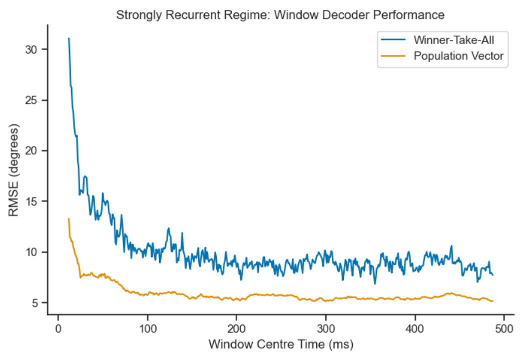


Fig 2.2.1(upper left): Performance of the winner-take-all and population vector decoders in the Hubel-Wiesel regime, using time windows.

Fig 2.2.2: Performance of the winner-take-all and population vector decoders in the Uniform Inhibition regime, using time windows.

Fig 2.2.3: Performance of the winner-take-all and population vector decoders in the Strongly Recurrent regime, using time windows.

Using time windows, winner-take-all consistently performs worse than the population vector decoder, having much higher RMSE values. For both decoding methods, performance generally improves as the window centre time increases. RMSE values are higher when using instantaneous windows than cumulative spiking, leading to less useful data in windows. Instantaneous windows also show greater fluctuations trial-to-trial versus the cumulative functions. Overall, the cumulative spikes method is much more accurate than the time window method. However, it is slower, as it is not sensitive to rapid stimulus changes like the window method. The Strongly Recurrent regime continuously outperforms the others in terms of having the lowest RMSE, as its high tuning characteristic filters noise (outside of preferred stimulus inhibited, close to preferred stimulus strengthened).

3. Interpretation and Implications

3.1 Optimal Decoders

This section illustrates that no network is fully optimal. This is because thresholds and nonlinearities distort input nonlinearly, correlations waste information and decoders only see spikes (not continuous rates). This can be shown in the three different regimes:

Hubel and Wiesel regime: As a feedforward only network, using the cumulative method with a population vector decoder is the best option. This option allows the network to integrate for 50ms and the decoder to work over 500ms, leading to near-optimal averaging. Using a winner take-all method is suboptimal as it discards information from non-winning neurons (in this regime many neurons contain information that is discarded). Additionally, instantaneous windows only have a 25ms window, which is insufficient for averaging. Using cumulative population vector decoding, however, still has suboptimalities, as there is a lack of efficiency (neurons are not sharply tuned, leading to less targeted information to decode).

Uniform Inhibition regime: This network uses aggressive thresholding, as only the preferred stimulus and close neurons do not have activity suppressed. Cumulative decoders perform the best in this regime as they have more information to average, versus small instantaneous windows. Population vector decoding still beats winner-take-all decoding as the former uses all available spikes. However, spikes are highly supressed in this regime, making any decoder in this network suboptimal (not enough spikes to get a stable estimate of a neuron's response).

Strongly Recurrent regime: This network inhibits neurons far from the preferred stimulus, while amplifying activity near it. This sharpens tuning and signal quality but increases correlations(Q3). It performs well despite the correlations as sharp tuning provides strong signals and stabilises the network, providing a reliable readout. As with the other regimes, population vector decoding with a cumulative readout is the most optimal (more input included in decoding). However, it is not as great of a difference here, as the network already does internal temporal integration. Furthermore, there is a source of suboptimality for the population vector decoding with a cumulative readout, as correlations (found at a high level in this regime) lead to redundancies in information, reducing effectiveness.

3.2 Different Approaches

Lectures in INFR11209 state that sharper tuning and weaker correlations provide the best coding. Additionally, the results from simulating a Strongly Recurrent network regime have the sharpest tuning but strong correlations. The SR network regime has the best decoding performance (Fig. 3.1.3, Fig. 3.2.3) but this is not purely because of sharpening. Fisher information (how precisely a stimulus can be estimated from neural responses) can be found using the formula $I = \Sigma_i (f'_i)^2 / (\sigma_i^2 + \text{correlation terms})$, where sharpening increases $(f'_i)^2$ but neural correlations with the same tuning increase the denominator. Therefore, the best network is determined by the balance between tuning sharpness and correlation costs.

In simple encoding models found in INFR11209's lectures, neurons are assumed as independent, with tuning shape and correlations as parameters independent of each other. This is mathematically simpler and identifies a theoretical optimum while showing clear predictions for good neural coding. However, it does not explain how to reach the optimal coding and ignores that connectivity weights creates both tuning and correlations. Recurrent models recognise this and cannot optimise independently (more tuning -> more correlations), revealing trade-offs between both factors. This model reveals constraints, is testable through small changes, and shows dynamics. However, it is more complex and computationally expensive, with conclusions depending on specific choices for the balance between tuning and correlations. Both systems are essential and complimentary, as encoding models tell us what's theoretically possible and biologically achievable given mechanistic constraints, and come to the same conclusion: **the best network coding is one where tuning curves are sharp and correlations are low.**

4. Appendix

4.1 Code

```python
# COURSEWORK: COMPUTATIONAL NEUROSCIENCE(INFR 11209)
# This code simulates different 3 different regimes for a Ring Network System, creating plots for Q1, Q2, Q3, Q4,and Q5

#===========================================================================================================================================

# IMPORT NECESSARY LIBRARIES

#numerical analysis, plotting
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

#styling
from scipy.ndimage import uniform_filter1d
import seaborn as sns
sns.set()
# global defaults for plot styling
sns.set_theme(style="ticks",
              palette="colorblind",
              font_scale=1.0,
              rc={
              "axes.spines.right": False,
              "axes.spines.top": False,
             },
             )

#===========================================================================================================================================

# SIMULATION FUNCTIONS

# Ensure non-negative values scaled by beta (threshold nonlinearity)
def phi(v, beta = 0.1):
    return beta * np.maximum(v, 0)

# Euler method for this ring model, predicting how something changes from one moment to the next
def euler(w, u_i, v_initial, N, sim_time, d_time, tau, beta, sigma):

    voltages = np.zeros((sim_time, N)) # initialises a vector to hold voltages of each neuron at each timestamp
    spikes = np.zeros((sim_time, N)) # initialises a vector to track the spikes of each neuron at each timestamp
    inputs = np.zeros((sim_time, N)) # initialises a vector to hold the inputs of each neuron at each timestamp
    v = v_initial.copy() # creates a copy of the initial voltage matrix

    #euler loop, updates the network one timestep at a time
    #records voltages, inputs, and spikes using a Poisson proccess
    for time in range(sim_time):
        q = np.random.randn(N) #random numbers in N(0,1) for each neuron
        r = phi(v, beta) #vector containing firing rates for each neuron

        #calculate noise inputs and store them
        noise_input = np.sqrt(d_time/tau) * sigma * q
        inputs[time] = u_i + noise_input

        dV = (-v + w @ r + u_i) * (d_time / tau) + noise_input #differential equation for Euler function
        v = v + dV

        #store values
        voltages[time] = v
        spikes[time] = np.random.poisson (r * d_time)

    return voltages, spikes, inputs

# Single trials: Q1
def plot_single_trials(voltages, spikes, inputs, beta, title):

    # setup space for 4 plots (4 outputs)
    fig, axs = plt.subplots(2,2, figsize = (14,8))

    #plot 1: heatmap of voltages V(t)
    sns.heatmap(voltages.T, ax=axs[0, 0])
    axs[0,0].set_title(f"{title} Voltages V(t)")
    axs[0,0].set_xlabel("Time (ms)")
    axs[0,0].set_ylabel("Neuron")

    #plot 2: heatmap of firing rate r(t)
    firing_rate = phi(voltages, beta)
    sns.heatmap(firing_rate.T, ax = axs[0, 1])
    axs[0,1].set_title(f"{title} Firing Rate r(t)")
    axs[0,1].set_xlabel("Time (ms)")
    axs[0,1].set_ylabel("Neuron")

    #plot 3: heatmap of input
    sns.heatmap(inputs.T, ax = axs[1,0])
    axs[1,0].set_title(f"{title} Input")
    axs[1,0].set_xlabel("Time (ms)")
    axs[1,0].set_ylabel("Neuron")

    #plot 2: raster plot of spikes
    spike_times, neuron_ids = np.nonzero(spikes)
    axs[1,1].scatter(spike_times, neuron_ids, s = 2, color = "black")
    axs[1,1].set_title(f"{title} Spiking Activity")
    axs[1,1].set_xlabel("Time (ms)")
    axs[1,1].set_ylabel("Neuron")

    plt.tight_layout() #ensure plots do not overlap
    plt.show()

    return

# Tuning curves (many trials): Q2

# compute the tuning curves
def compute_tuning_curves(w, u_i, v_initial, N, sim_time, d_time, tau, beta, sigma, trials = 50, smooth_window = 10):

    tuning = np.zeros((sim_time, N)) #tuning = matrix storing spike counts for each neuron

    #create fixed random seed for reproducibility
    for seed in range(trials):
        np.random.seed(seed)
        _, spikes, _ = euler(w, u_i, v_initial, N, sim_time, d_time, tau, beta, sigma)
        tuning += spikes

    tuning /= (trials * d_time) # avg firing rate/ms

    #smoothen the tuning curve
    if smooth_window > 1:
        tuning = uniform_filter1d(tuning, size = smooth_window, axis = 0, mode = "nearest")

    return tuning

```

```python
# plotting the tuning curves
def plot_tuning_curves(tuning, s_i, title, times = (50, 200, 400)):

    plt.figure(figsize = (10,6))

    for t in times:
        plt.plot(s_i, tuning[t], label = f"t = {t} ms")

    plt.title(f"{title}: Population tuning curves")
    plt.xlabel("Preferred orientation (rad)")
    plt.ylabel("Firing rate(spikes/ms)")
    plt.legend()
    plt.show()

    return

# Noise correlations: Q3

#compute the noise correlations
def compute_noise_correlations(w, u_i, v_initial, N, sim_time, d_time, tau, beta, sigma, trials = 50, start = 400, window = 100):

    #initialise spike counts to 0
    spike_counts = np.zeros((trials, N))

    #create fixed random seed for reproducibility
    for seed in range(trials):
        np.random.seed(seed)
        _, spikes, _ = euler(w, u_i, v_initial, N, sim_time, d_time, tau, beta, sigma)
        spike_counts[seed] = np.sum(spikes[start: start + window], axis = 0)

    #matrix showing covariance (neuron correlation)
    sigma_matrix = np.cov(spike_counts, rowvar = False)

    return sigma_matrix

#plot noise correlations
def plot_noise_correlations(sigma_matrix, title):

    plt.figure(figsize = (8, 6))
    sns.heatmap(sigma_matrix, center = 0)
    plt.title(f"{title} Noise Correlations")
    plt.xlabel("Neuron")
```

```python
        plt.ylabel("Neuron")
        plt.show()

        return

    #=======================================================================================================================================

    #DECODING FUNCTIONS

    # decode the stimulus using the winner take all method
    def winner_take_all_decoder (spike_counts, preferred_orientations):

        #find the neuron with the maximum spike count, return the preferred orientation of the most active neuron
        winner = np.argmax(spike_counts)
        decoded_angle = preferred_orientations[winner]

        return decoded_angle

    # decode the stimulus using the population vector method
    def population_vector_decoder(spike_counts, preferred_orientations):

        #convert preferred orientations to unit vectors
        vector_x = np.cos(preferred_orientations) * spike_counts
        vector_y = np.sin(preferred_orientations) * spike_counts

        #sum up the vectors
        sum_x = np.sum(vector_x)
        sum_y = np.sum(vector_y)

        #decode the angle
        decoded_angle = np.arctan2(sum_y, sum_x)

        #return the decoded angle
        return decoded_angle

    # Q4: Cumulative Decoder Analysis (applies decoders to cumulative spike counts)

    #analysis
    def cumulative_decoder_analysis(w, u_i, v_initial, N, sim_time, d_time, tau, beta, sigma,
                                    true_stimulus = 0, trials = 50):

```

```python
        #set up neurons to be equally spaced out in the ring
        preferred_orientations = np.linspace(-np.pi, np.pi, N, endpoint = False)

        #storage for decoded values
        wta = np.zeros((trials, sim_time)) #winner take all value storage
        pv = np.zeros((trials, sim_time)) #population vector value storage

        for trial in range(trials):
            np.random.seed(trial)
            _, spikes, _ = euler (w, u_i, v_initial, N, sim_time, d_time, tau, beta, sigma)

            #cumulative spike counts at each time stamp
            cum_spikes = np.cumsum(spikes, axis = 0)

            for time in range(sim_time):
                spike_counts = cum_spikes[time, :]

                #skip if there are no spikes
                if np.sum(spike_counts) == 0:
                    pv[trial, time] = np.nan
                    wta[trial, time] = np.nan
                    continue

                #decode (call decoding functions)
                wta[trial, time] = winner_take_all_decoder(spike_counts, preferred_orientations)
                pv[trial, time] = population_vector_decoder(spike_counts, preferred_orientations)

        #compute the root mean square error at each time point
        wta_rmse = np.zeros(sim_time)
        pv_rmse = np.zeros(sim_time)

        for time in range(sim_time):

            #remove NaN trials for this timestamp
            wta_valid = wta[:, time][~np.isnan(wta[:, time])]
            pv_valid = pv[:, time][~np.isnan(pv[:, time])]

            #compute errors for PV and WTA

            #WTA
            if len(wta_valid) > 0:
                wta_errors = np.angle(np.exp(1j * (wta_valid - true_stimulus)))
                wta_rmse[time] = np.sqrt(np.mean(wta_errors ** 2))
            else:
                wta_rmse[time] = np.nan

            #PV
            if len(pv_valid) > 0:
                pv_errors = np.angle(np.exp(1j * (pv_valid - true_stimulus)))
                pv_rmse[time] = np.sqrt(np.mean(pv_errors ** 2))
            else:
                pv_rmse[time] = np.nan

        #convert output to degrees and return
        wta_rmse_degrees = np.rad2deg(wta_rmse)
        pv_rmse_degrees = np.rad2deg(pv_rmse)
        return wta_rmse_degrees, pv_rmse_degrees

    #plotting Q4
    def plot_cumulative_rmse(time, wta_rmse, pv_rmse, title):
        plt.figure(figsize = (8,5))
        plt.plot(time, wta_rmse, label = "Winner-Take-All")
        plt.plot(time, pv_rmse, label = "Population Vector")
        plt.xlabel("Time (ms)")
        plt.ylabel("RMSE (degrees)")
        plt.title(f"{title}: Cumulative Decoder Performance")
        plt.legend()
        plt.show()

        return

    #Q5: Spiking Activities Within Time Windows

    #analysis
    def window_decoder_analysis(w, u_i, v_initial, N, sim_time, d_time, tau, beta, sigma,
                                true_stimulus = 0, trials = 50, window_size = 25):

        #set up neurons to be equally spaced out in the ring
        preferred_orientations = np.linspace(-np.pi, np.pi, N, endpoint = False)

        #window centres
        half_window = window_size // 2
        window_centres = np.arange(half_window, sim_time - half_window)

        wta = np.zeros((trials, len(window_centres)))
        pv = np.zeros((trials, len(window_centres)))

        for trial in range(trials):
            np.random.seed(trial)
            _,spikes,_ = euler(w, u_i, v_initial, N, sim_time, d_time, tau, beta, sigma)

            for i, tc in enumerate(window_centres):
                t_start = tc - half_window
                t_end = tc + half_window

                # sum spikes in window from t_start to t_end
                spike_counts = np.sum(spikes[t_start:t_end, :], axis = 0)

                #skip if there are no spikes
                if np.sum(spike_counts) == 0:
                    wta[trial, i] = np.nan
                    pv[trial, i] = np.nan
                    continue

                wta[trial, i] = winner_take_all_decoder(spike_counts, preferred_orientations)
                pv[trial, i] = population_vector_decoder(spike_counts, preferred_orientations)

        #compute root mean square error for each window centre
        wta_rmse = np.zeros(len(window_centres))
        pv_rmse = np.zeros(len(window_centres))

        for i in range(len(window_centres)):
            wta_valid = wta[:, i][~np.isnan(wta[:, i])]
            pv_valid = pv[:, i][~np.isnan(pv[:, i])]

            if len(wta_valid) > 0:
                wta_errors = np.angle(np.exp(1j * (wta_valid - true_stimulus)))
                wta_rmse[i] = np.sqrt(np.mean(wta_errors ** 2))
            else:
```

```python
            wta_rmse[i] = np.nan

        if len(pv_valid) > 0:
            pv_errors = np.angle(np.exp(1j * (pv_valid - true_stimulus)))
            pv_rmse[i] = np.sqrt(np.mean(pv_errors ** 2))
        else:
            pv_rmse[i] = np.nan

    wta_rmse_degrees = np.rad2deg(wta_rmse)
    pv_rmse_degrees = np.rad2deg(pv_rmse)

    return window_centres, wta_rmse_degrees, pv_rmse_degrees

#plotting
def plot_window_rmse(window_centres, wta_rmse, pv_rmse, title):
    plt.figure(figsize = (8,5))
    plt.plot(window_centres, wta_rmse, label = "Winner-Take-All")
    plt.plot(window_centres, pv_rmse, label = "Population Vector")
    plt.xlabel("Window Centre Time (ms)")
    plt.ylabel("RMSE (degrees)")
    plt.title(f"{title}: Window Decoder Performance")
    plt.legend()
    plt.show()

    return

#======================================================================================================================================

#MAIN FUNCTION TO SIMULATE AND PLOT VALUES

def main():

    #delcare variables/parameters
    S = 0 #stimulus
    N = 100 #100 neurons
    D_time = 1 #timestep = 1ms
    Sim_time = 500 #simulation time = 500 ms
    V_initial = np.zeros(N)  #voltage of any neuron "i" starts at 0
    Tau = 50 #time constant for system change
    Beta = 0.1 #the non-linearality parameter (controlling how the neuron’s firing rate is influenced by its membrane potential)
    Sigma = 0.1 # noise affecting the neurons
    U0 = 0.5 #initial neuron stimulus
```

```
        U1 = 0.5 #additional neuron stimulus

        #setting up simulation space
        S_i = np.linspace(-np.pi, np.pi, N, endpoint = False) # setting up each neuron's orientation in the simulation (evenly spaced in ring)

        #average feedforward input for the stimulus orientation
        U_i = U0 + (U1 * np.cos(S - S_i))

        # connection matrices for specific simulations
        W_hubel_wiesel = 0 + 0 * np.cos(np.subtract.outer(S_i, S_i))
        W_uniform_inhibition = -4 + 0 * np.cos(np.subtract.outer(S_i, S_i))
        W_recurrent = -10 + 11 * np.cos(np.subtract.outer(S_i, S_i))

        #find values for voltages, spikes, and inputs to be plotted
        voltages_hw, spikes_hw, inputs_hw = euler(W_hubel_wiesel, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma)
        voltages_ui, spikes_ui, inputs_ui = euler(W_uniform_inhibition, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma)
        voltages_re, spikes_re, inputs_re = euler(W_recurrent, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma)

        #plotting single trial simulations(Q1)
        plot_single_trials(voltages_hw, spikes_hw, inputs_hw, Beta, "Hubel and Wiesel Regime")
        plot_single_trials(voltages_ui, spikes_ui, inputs_ui, Beta, "Uniform Inhibition Regime")
        plot_single_trials(voltages_re, spikes_re, inputs_re, Beta, "Strongly Recurrent Regime")

        #computing tuning curves(Q2)
        tuning_hw = compute_tuning_curves(W_hubel_wiesel, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma)
        tuning_ui = compute_tuning_curves(W_uniform_inhibition, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma)
        tuning_re = compute_tuning_curves(W_recurrent, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma)

        plot_tuning_curves(tuning_hw, S_i, "Hubel and Wiesel")
        plot_tuning_curves(tuning_ui, S_i, "Uniform Inhibition Regime")
        plot_tuning_curves(tuning_re, S_i, "Strongly Recurrent Regime")

        #plotting noise correlations(Q3)
```

```python
        sigma_matrix_hw = compute_noise_correlations(W_hubel_wiesel, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma)
        sigma_matrix_ui = compute_noise_correlations(W_uniform_inhibition, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma)
        sigma_matrix_re = compute_noise_correlations(W_recurrent, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma)

        plot_noise_correlations(sigma_matrix_hw, "Hubel and Wiesel")
        plot_noise_correlations(sigma_matrix_ui, "Uniform Inhibition Regime")
        plot_noise_correlations(sigma_matrix_re, "Strongly Recurrent Regime")

        #apply decoders using cumulative spike counts (Q4)
        wta_rmse_hw_cum, pv_rmse_hw_cum = cumulative_decoder_analysis(W_hubel_wiesel, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma,
                                                                                  true_stimulus = 0, trials = 50)
        wta_rmse_ui_cum, pv_rmse_ui_cum = cumulative_decoder_analysis(W_uniform_inhibition, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma,
                                                                                  true_stimulus = 0, trials = 50)
        wta_rmse_re_cum, pv_rmse_re_cum = cumulative_decoder_analysis(W_recurrent, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma,
                                                                                  true_stimulus = 0, trials = 50)

        plot_cumulative_rmse(np.arange(Sim_time), wta_rmse_hw_cum, pv_rmse_hw_cum, "Hubel and Weisel Regime")
        plot_cumulative_rmse(np.arange(Sim_time), wta_rmse_ui_cum, pv_rmse_ui_cum, "Uniform Inhibition Regime")
        plot_cumulative_rmse(np.arange(Sim_time), wta_rmse_re_cum, pv_rmse_re_cum, "Strongly Recurrent Regime")

        #apply decoders using time windows (Q5)
        window_centres_hw, wta_rmse_hw_win, pv_rmse_hw_win = window_decoder_analysis(W_hubel_wiesel, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma,
                                                                                  true_stimulus = 0, trials = 50)
        window_centres_ui, wta_rmse_ui_win, pv_rmse_ui_win = window_decoder_analysis(W_uniform_inhibition, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma,
                                                                  true_stimulus = 0, trials = 50)
        window_centres_re, wta_rmse_re_win, pv_rmse_re_win = window_decoder_analysis(W_recurrent, U_i, V_initial, N, Sim_time, D_time, Tau, Beta, Sigma,
                                                                  true_stimulus = 0, trials = 50)

        plot_window_rmse(window_centres_hw, wta_rmse_hw_win, pv_rmse_hw_win, "Hubel and Weisel Regime")
        plot_window_rmse(window_centres_ui, wta_rmse_ui_win, pv_rmse_ui_win, "Uniform Inhibition Regime")
        plot_window_rmse(window_centres_re, wta_rmse_re_win, pv_rmse_re_win, "Strongly Recurrent Regime")

        return

    #ensure program calls main() when run
    if __name__ == "__main__":
        main()

```