

Module	4G3	Title of report	COMPUTATIONAL NEUROSCIENCE A2: RECURRENT NEURAL NETWORK DYNAMICS		
Date submitted:		4/11/2025	Assessment for this module is <input checked="" type="checkbox"/> 100% / <input type="checkbox"/> 25% coursework of which this assignment forms <u>50%</u> %		
UNDERGRADUATE and POST GRADUATE STUDENTS					
Name:	Mac walker	College:	MAGDALENE	<input type="checkbox"/> Undergraduate <input checked="" type="checkbox"/> Post graduate	

Feedback to the student		Very good	Good	Needs improvmt	
<input type="checkbox"/> See also comments in the text					
C O N T E N T	Completeness, quantity of content: Has the report covered all aspects of the lab? Has the analysis been carried out thoroughly?				
	Correctness, quality of content Is the data correct? Is the analysis of the data correct? Are the conclusions correct?				
	Depth of understanding, quality of discussion Does the report show a good technical understanding? Have all the relevant conclusions been drawn?				
	Comments:				
P R E S E N T A T I O N	Attention to detail, typesetting and typographical errors Is the report free of typographical errors? Are the figures/tables/references presented professionally?				
	Comments:				

1 Question 1

After integrating the 4 models described in the Appendix, we show the populating tuning curves in Figure 1.

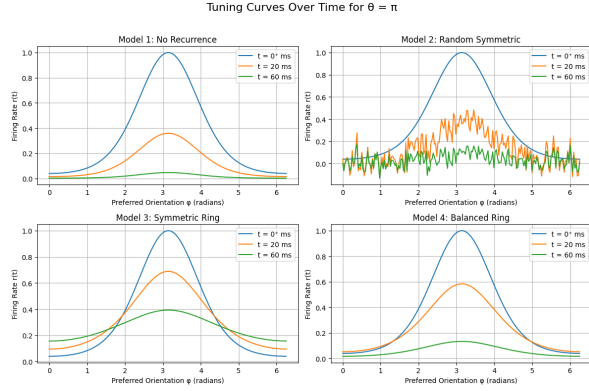


Figure 1: V1 Population Tuning Curves at $t = 0^+, 20, 60$ ms (Note: all images may be found enlarged in the Appendix)

2 Question 2

In general, after the initial stimulus, the dynamics of the network evolve according to the following first-order differential equation:

$$\tau \frac{dr}{dt} = -r + Wr$$

Rearranging, we get:

$$\frac{dr}{dt} = \frac{1}{\tau}(-I + W)r = \frac{1}{\tau}Ar$$

where we define:

$$A := W - I$$

To understand the evolution of the population activity $r(t)$, we study the properties of the matrix A , which governs the dynamics of the system.

Let A have an eigendecomposition:

$$A = Q\Lambda Q^{-1}$$

where Λ is a diagonal matrix of eigenvalues λ_i , and Q contains the corresponding eigenvectors v_i as columns. Then the solution to the system is:

$$r(t) = \sum_i c_i e^{\frac{\lambda_i t}{\tau}} v_i$$

for coefficients c_i determined by the initial condition $r(0)$. Note that since $A = W - I$, we have:

$$\lambda_i^A = \lambda_i^W - 1$$

so eigenvalues of A reflect how far the eigenvalues of W are from 1.

Spectral Properties of W_2

Model 2's weight matrix W_2 is constructed as a random symmetric matrix. According to random matrix theory, the eigenvalues of such matrices typically follow the Wigner semicircle distribution, and the corresponding eigenvectors are spatially unstructured [1]. The input $h(\theta)$ is a smooth von Mises function, that projects onto the eigenvectors of W_2 . Each mode evolves as $e^{\lambda_i t / \tau}$, but if no λ_i is large and aligned with $h(\theta)$, the system cannot reinforce a coherent population code. The apparent stochasticity we see is due to the eigenvectors which are spatially unaligned with the input. This causes the system to evolve through a noisy blend of weak, incoherent modes, resulting in jagged activity across neurons and time. Figure 2 displays the top 3 (ordered by $\text{Re}(\lambda)$) eigenvectors for $b=W_2$ and W_3 , showing the inherently jagged and spatially

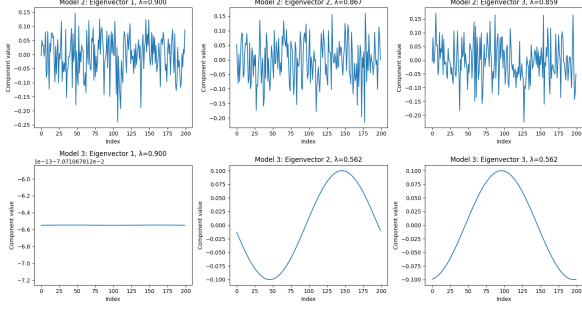


Figure 2: Top 3 Eigenvectors of W_2 and W_3

unstructured nature of the eigenvectors.

3 Question 3

In Model 3, the recurrent matrix W_3 has a ring structure and its leading eigenvectors are smooth and aligned with the orientation-tuned input h_θ . This means that the components aligned with large eigenvalues grow coherently over time. This can be seen in Figure 2.

In contrast, Model 2's W_2 is a random symmetric matrix. Its eigenvectors are spatially unstructured and the input does not align well with any dominant mode. As a result, the activity in Model 2 is spread over many incoherent directions and is less amplified.

Therefore, Model 3 exhibits stronger, more coherent population activity at 60 ms, which can be seen in Figure 1.

4 Question 4: Decoding Accuracy Across Models

When analysing the decoding error over time for each model shown in Figure 3, we

observe the following:

- **Model 1** shows a steady increase in decoding error as the stimulus-driven activity decays. Without recurrent input, the representation fades over time.
- **Model 2** performs poorly throughout due to its random symmetric connectivity. The lack of structured recurrence leads to noisy spatial activity patterns, which degrade decoding performance.
- **Model 3** maintains significantly better accuracy than Models 1 and 2, particularly at later time points. Its ring-structured connectivity reinforces the orientation code in a spatially coherent way.
- **Model 4** performs better than Models 1 and 2 with its balanced excitatory-inhibitory structure, though it does not outperform Model 3 under default parameters.

These results confirm that models with organised eigenstructure (Models 3 and 4) preserve and amplify input-aligned modes, supporting accurate decoding over time. In contrast, random connectivity in Model 2 leads to noisy, incoherent activity due to unstructured eigenvectors, while the absence of recurrence in Model 1 causes signal decay and degraded performance.

5 Question 5

As seen in Figure 3, increasing α' , the spectral radius of the recurrent weight matrix in

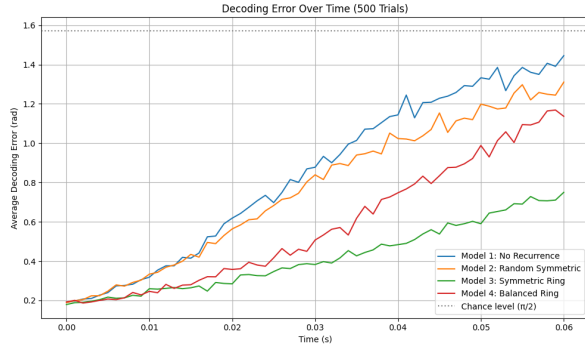


Figure 3: Decoding Error over Time for all 4 Models

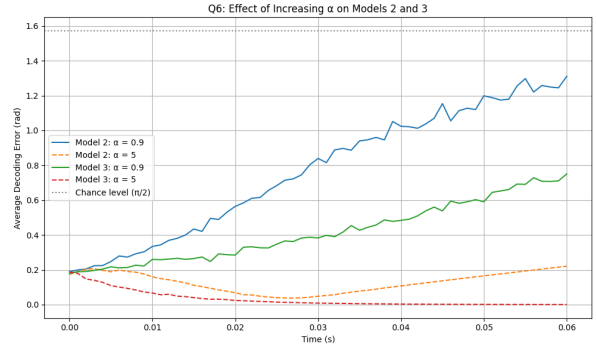


Figure 5: Decoding Error for W_2 and W_3 with different α'

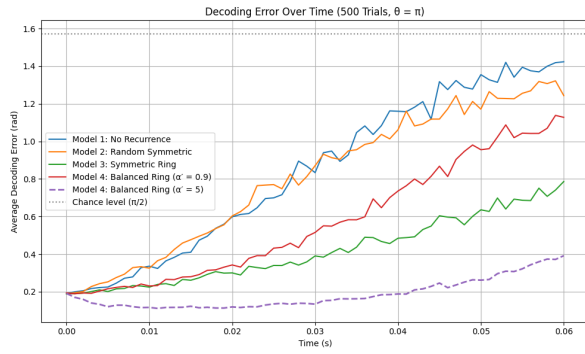


Figure 4: Decoding Error over time for 4 models, with varying α' in Model 4

Model 4, leads to improved decoding performance. As the effective matrix $A = W - I$ becomes more positively weighted (i.e., larger eigenvalues), the dominant modes of the system grow more rapidly over time. Increasing α' therefore leads to stronger amplification of the stimulus-aligned mode, improving the signal-to-noise ratio and decoding fidelity.

Model 4 includes an inhibitory subpopulation, which provides feedback that limits runaway excitation. This will be discussed further in Question 8.

6 Question 6

Inspecting Figure 5, it would seem as though we have a 'free lunch', whereby in-

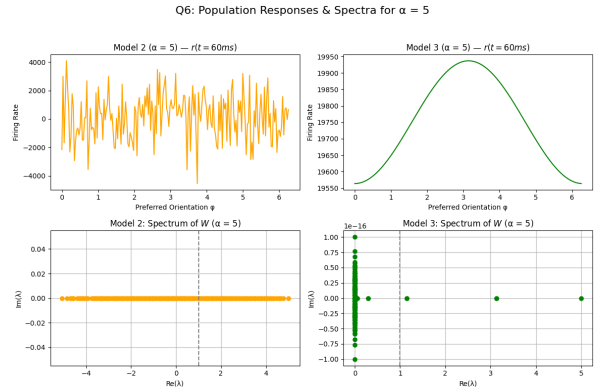


Figure 6: Population Response and Spectra for $\alpha = 5$ in Model 2 and 3

creasing α' gives us decreasing decoding error.

However, when we consider Figure 6, we see that as we increase α' , the firing rate becomes entirely biologically implausible. This is due to the exponential growth of the firing rates of the neurons as the simulation develops, due to $\text{Re}(\lambda)$ being greater than 1.

7 Question 7

In the default implementation of Model 4, only the excitatory subpopulation receives feedforward input, with the inhibitory population left unmodulated:

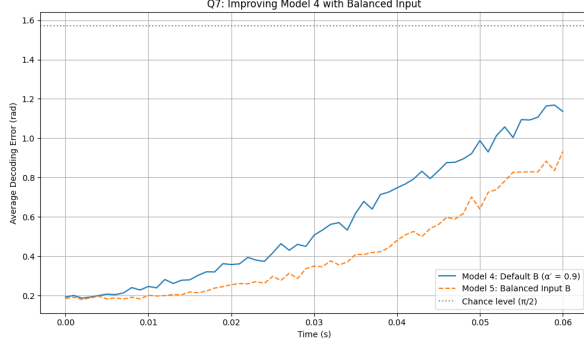


Figure 7: Improving Model 4 with Balanced Input

$$B = \begin{bmatrix} \mathbb{I}_m \\ \mathbf{0}_{m \times m} \end{bmatrix}$$

A biologically motivated modification, in which the inhibitory population receives an equal-magnitude but opposite-polarity input, is shown below:

$$B = \begin{bmatrix} \mathbb{I}_m \\ -\mathbb{I}_m \end{bmatrix}$$

This structure reflects cortical motifs observed in balanced networks, where excitation and inhibition are co-tuned but oppositely modulated.

As can be seen in Figure 7, the decoding error decreases with this new choice of feedforward input. We interpret this as evidence that engaging the inhibitory network via structured input improves the fidelity and speed of orientation coding.

8 Question 8

As α' increases, the recurrent network becomes increasingly selective, ultimately behaving like a hard-coded lookup table. While this sharpens tuning curves and improves decoding accuracy under noise-free

conditions, it reduces the ability of the network to decode under noisy conditions. Further, the resulting tuning parameters become implausibly narrow and discontinuous.

9 Question 9

Our results suggest that excitation-inhibition (E/I) balance plays a key role in the speed and precision of orientation coding in V1. Model 4, which includes a balanced inhibitory population, achieves high decoding accuracy while maintaining biologically plausible dynamics. Structured inhibition sharpens tuning, reduces noise, and stabilises activity even in high-gain regimes. This mirrors cortical motifs where excitation and inhibition are co-tuned, enabling fast and reliable sensory processing. In contrast, unbalanced or random connectivity leads to poor decoding or unstable growth. Overall, E/I balance appears essential for robust and efficient coding in recurrent cortical circuits.

References

- [1] Eugene P. Wigner. Random matrices in physics. *SIAM Review*, 9(1):1–23, 1967.

10 Appendix

Appendix Figures

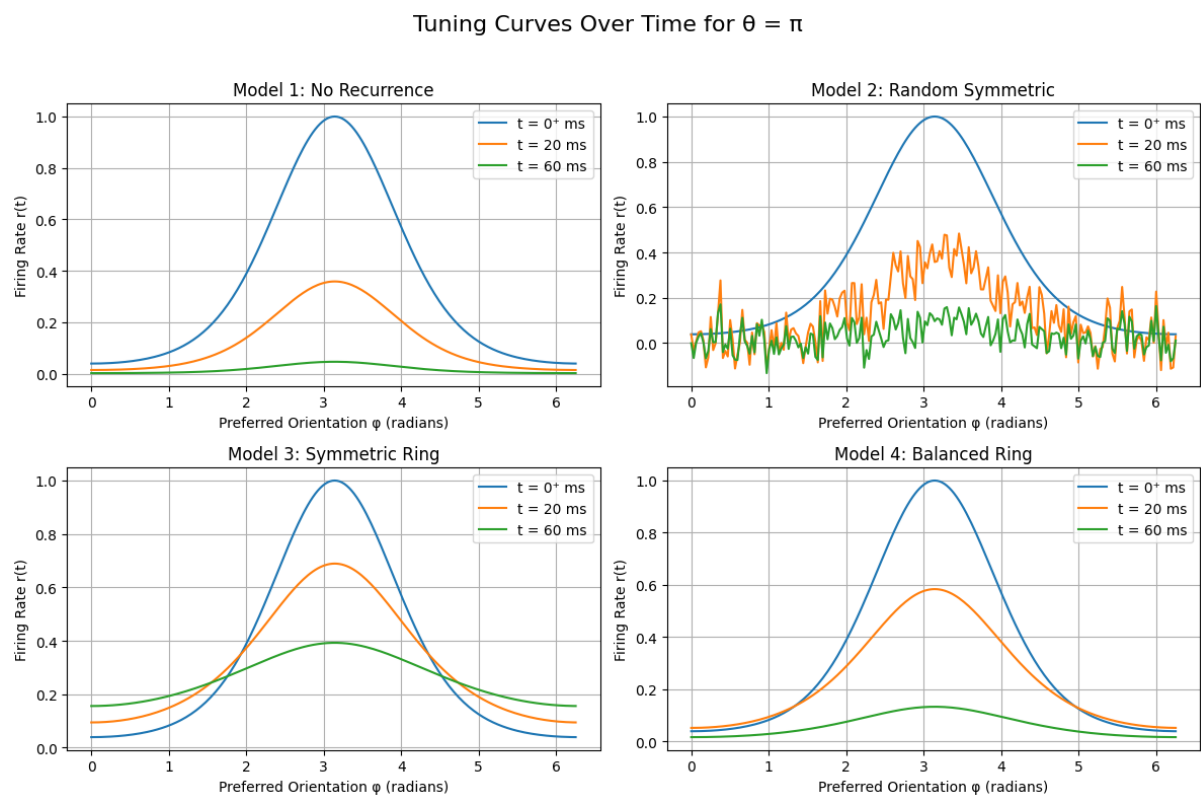


Figure 8: (Appendix) Full-size: V1 Population Tuning Curves at $t = 0^+$, 20, 60 ms

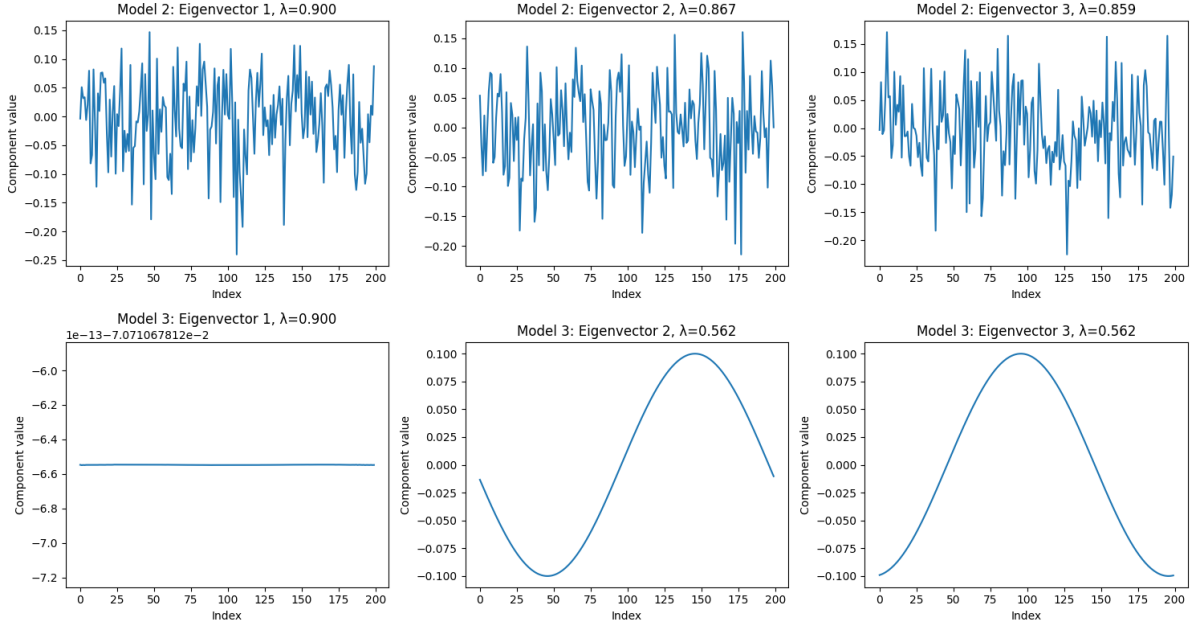


Figure 9: (Appendix) Full-size: Top 3 Eigenvectors of W_2 and W_3

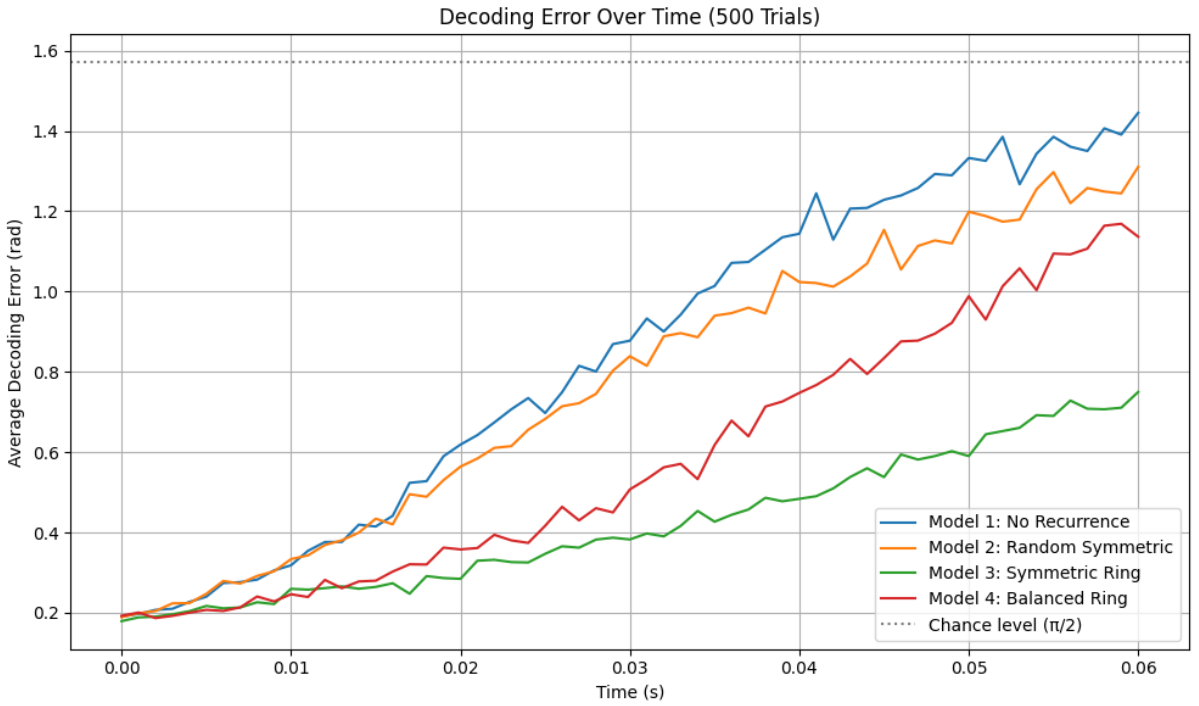


Figure 10: (Appendix) Full-size: Decoding Error over Time for all 4 Models

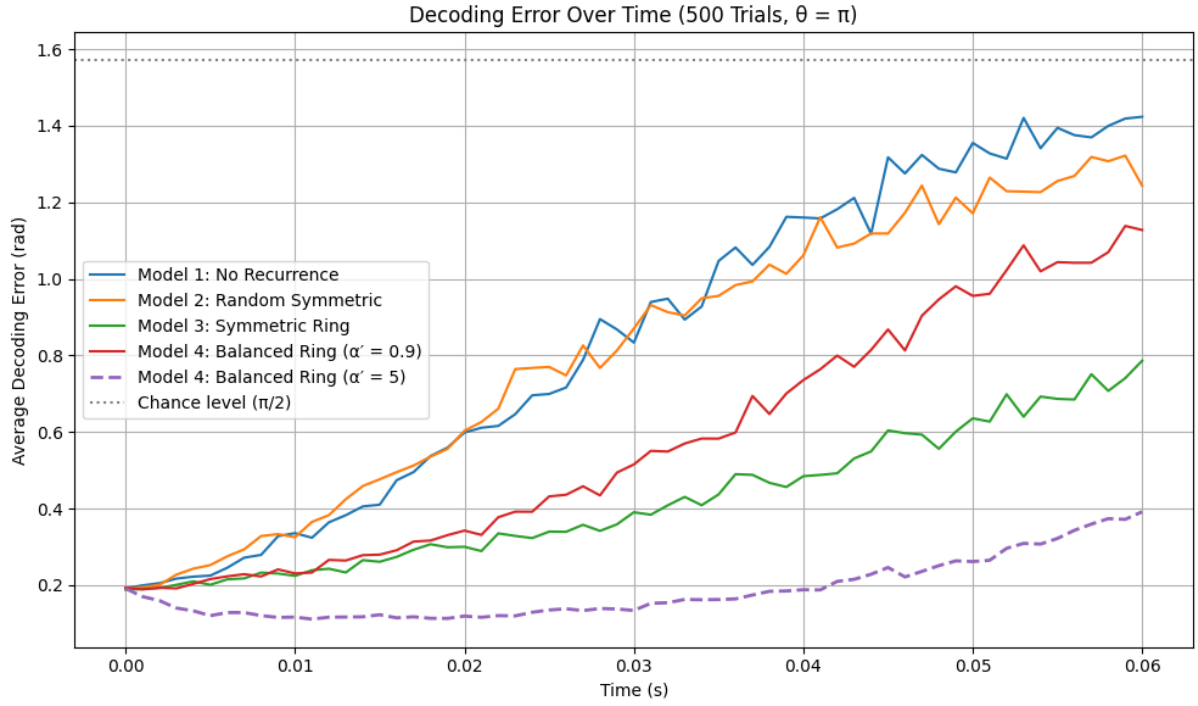


Figure 11: (Appendix) Full-size: Decoding Error for Model 4 at Different α'

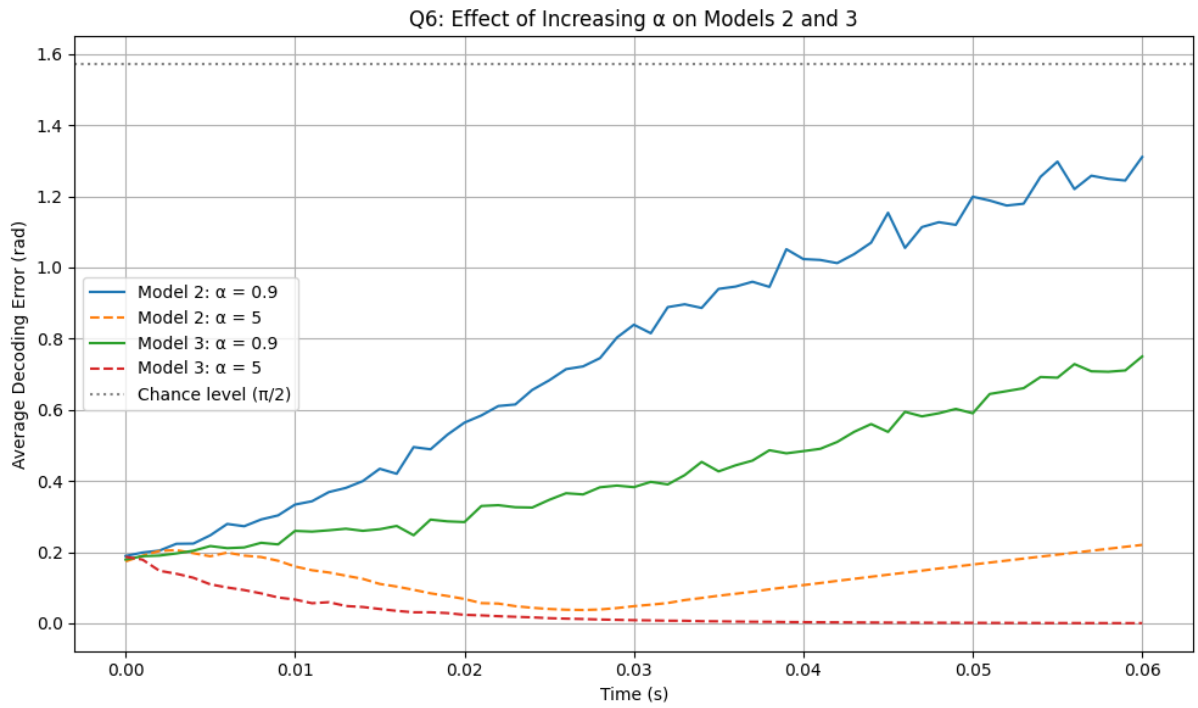


Figure 12: (Appendix) Full-size: Decoding Error for W_2 and W_3 with varying α

Q6: Population Responses & Spectra for $\alpha = 5$

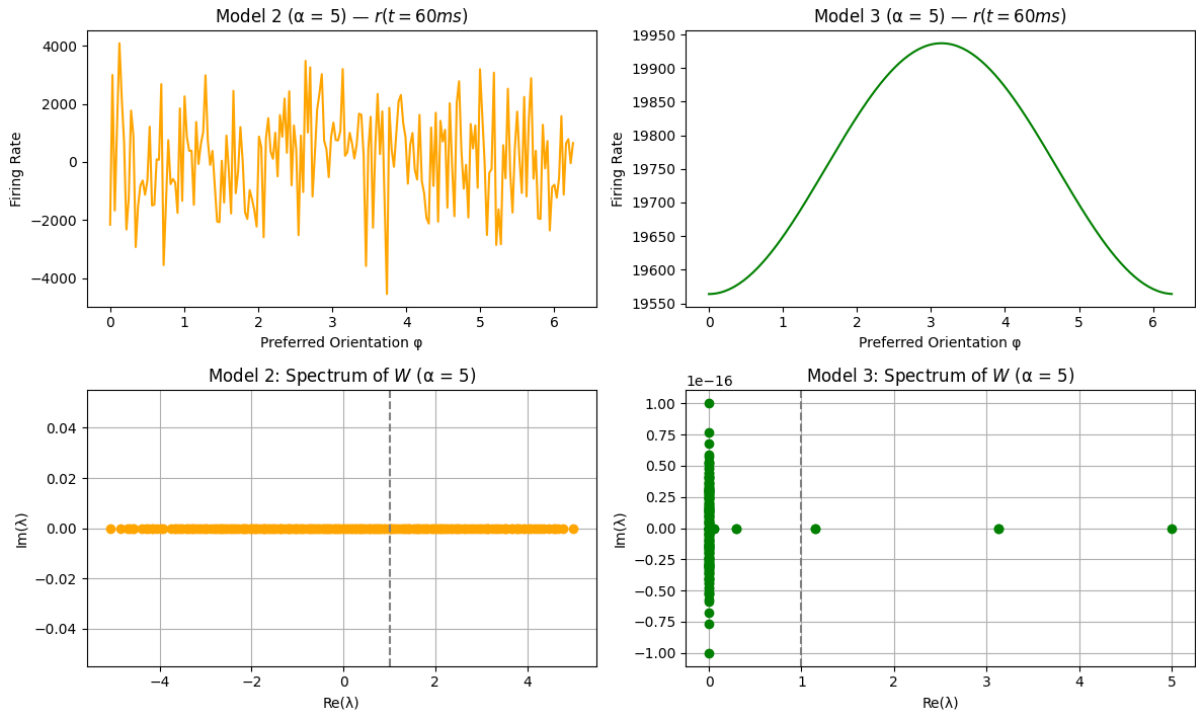


Figure 13: (Appendix) Full-size: Population Response and Spectra at $\alpha = 5$

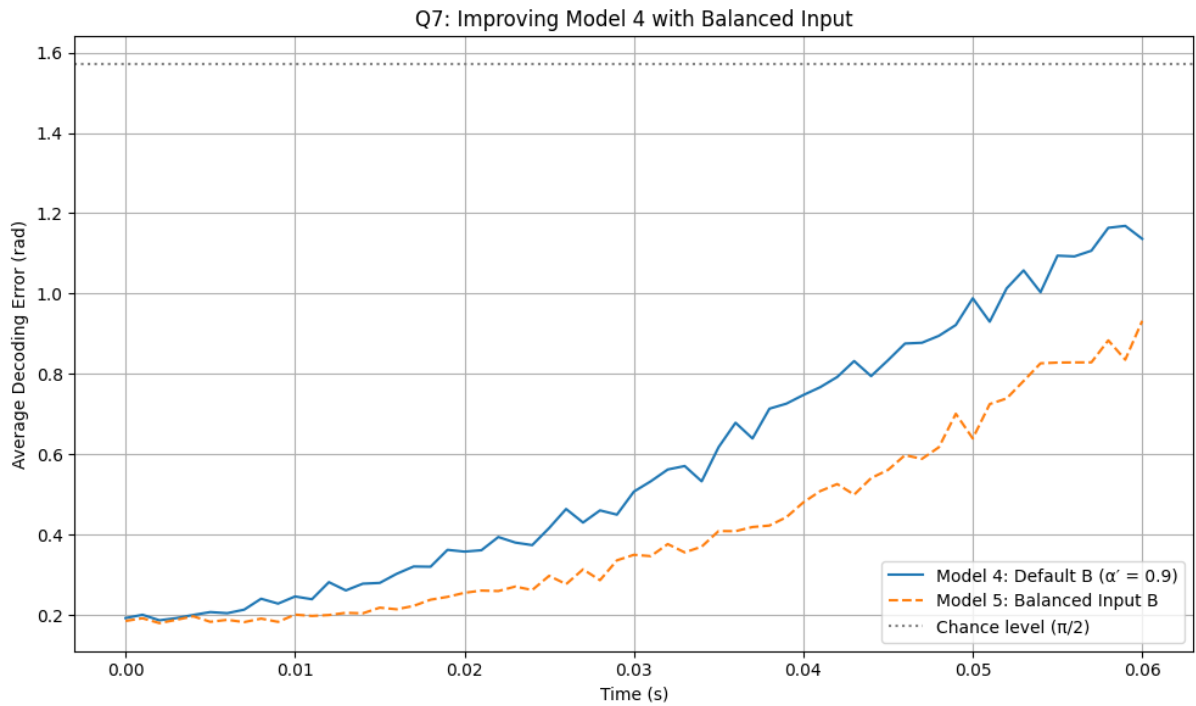


Figure 14: (Appendix) Full-size: Effect of Balanced Input in Model 4

11 Detailed Summary of Experiments

11.1 Simulation Setup

We define a grid of orientations $\phi \in [0, 2\pi)$ with $m = 200$ evenly spaced values. The number of neurons n is model-dependent. We use a tuning function $V(z) = \exp\left(\frac{\cos(z)-1}{k^2}\right)$ to define input selectivity, where $k = \pi/4$ sets the tuning width. The encoding of a stimulus at orientation θ is given by $h(\theta) = V(\phi - \theta)$.

The recurrent dynamics are updated using Euler discretisation:

$$r_{t+1} = r_t + \frac{dt}{\tau} \left(-r_t + W r_t + \delta_{t,0} \frac{1}{dt} B h(\theta) \right)$$

Readout is defined as a noisy linear transformation: $\tilde{o}(t) = C r(t) + \sigma \varepsilon$, where ε is Gaussian noise. The decoded orientation estimate $\hat{\theta}$ is obtained via a circular decoder.

We simulate the network response at time steps $t = 0, 20, 60$ ms, given an input $\theta = \pi$.

11.2 Model 1: No Recurrence

This model removes recurrence entirely by setting $W^{(1)} = 0_{m \times m}$. As expected, activity from the initial input decays uniformly without additional structure. Neuron responses are limited in both magnitude and temporal persistence, leading to low complexity in temporal dynamics. Visualisations confirm this passive decay, and the heatmap of population activity is smooth and Gaussian-like.

11.3 Model 2: Random Symmetric Connectivity

Here, recurrent weights are defined as $W^{(2)} = \mathcal{R}(\tilde{W} + \tilde{W}^\top, \alpha)$ with i.i.d. entries in $\tilde{W} \sim \mathcal{N}(0, 1)$. The matrix is then rescaled to have spectral radius $\alpha = 0.9$.

Compared to Model 1, the random symmetric connectivity introduces irregularity into the activity patterns. Although symmetry ensures a real eigenspectrum, the eigenvectors are misaligned with the input bump, leading to amplified noisy modes and more complex, less structured temporal dynamics. These observations are supported by higher residuals in Gaussian fits over time and increasing noise in population responses.

11.4 Model 3: Symmetric Ring Structure

In this model, we define $W^{(3)} = \mathcal{R}(\tilde{W}, \alpha)$ where $\tilde{W}_{ij} = V(\phi_i - \phi_j)$. This introduces a structured ring attractor model, where connection strength depends on preferred orientation similarity.

The structured connectivity leads to sustained and smooth population activity centred around the input orientation. Temporal dynamics preserve the spatial structure of the

input, which allows for more reliable orientation decoding. Residuals from Gaussian fits remain low over time, reflecting coherent and noise-suppressed dynamics.

11.5 Model 4: Balanced Ring Structure

The most complex model doubles the number of neurons ($n = 2m$), implementing a balanced ring attractor with excitatory-inhibitory symmetry:

$$W^{(4)} = \begin{pmatrix} \tilde{W} & -\tilde{W} \\ \tilde{W} & -\tilde{W} \end{pmatrix}, \quad B = \begin{pmatrix} I_m \\ 0 \end{pmatrix},$$

$$C = \begin{pmatrix} I_m & 0 \end{pmatrix}$$

The balanced dynamics further reduce noise, stabilising the bump of activity without overshooting or amplifying irregular modes. The neuronal activity remains structured and consistent across time steps, and the low residuals validate this coherence.

12 Use of Generative AI

I used Generative AI throughout this assignment. The coding and programming was originally done by myself, with ChatGPT and Claude assistance only by interacting with this system as a tutor (appending all prompts with 'You are a socratic tutor - please don't give me the answer, just guide me to it' (I would *hugely* recommend this system - it's great!)). After getting stuck with the decoding function, I pair-debugged with ChatGPT, arriving at an updated script for my simulation and decoding functions. I then used this script for the latter questions (seen in Notebook - with Changes). With regards to the plotting - I haven't written matplotlib script in the last 3 months, as Generative AI has done all of it for me. Similarly, for iterating through simulations, I just asked ChatGPT to provide the scripts for these. GenerativeAI also helped me in writing my report, with LaTeX formatting and shortening of phrasing.

12.1 Plotting Prompt

Prompt: Given this function logic ... , please provide me with the matplotlib code to run through 100 simulations of this code for the 4 different models, and plot these all in one plot.

Model Used: ChatGPT, 4o [10/04/2025]

Output:

```

1 # --- Run everything for Model 1 and Model 2 ---
2 time_axis = np.linspace(0, duration, num_time_steps)
3 h_theta = h_theta_vector(phi, theta)

```

```

4
5 # --- Run everything for Models 1 to 4 ---
6 time_axis = np.linspace(0, duration, num_time_steps)
7 h_theta = h_theta_vector(phi, theta)
8
9 # Model 1
10 W1, B1, C1, _ = network_model(1, m, phi, alpha, alpha_prime)
11 r1 = simulate_model(W1, B1, C1, h_theta, tau, dt, num_time_steps)
12 error1 = compute_decoding_error(r1, C1, sigma, phi, theta,
    num_trials=500)
13
14 # Model 2
15 W2, B2, C2, _ = network_model(2, m, phi, alpha, alpha_prime)
16 r2 = simulate_model(W2, B2, C2, h_theta, tau, dt, num_time_steps)
17 error2 = compute_decoding_error(r2, C2, sigma, phi, theta,
    num_trials=500)
18
19 # Model 3
20 W3, B3, C3, _ = network_model(3, m, phi, alpha, alpha_prime)
21 r3 = simulate_model(W3, B3, C3, h_theta, tau, dt, num_time_steps)
22 error3 = compute_decoding_error(r3, C3, sigma, phi, theta,
    num_trials=500)
23
24 # Model 4
25 W4, B4, C4, _ = network_model(4, m, phi, alpha, alpha_prime)
26 r4 = simulate_model(W4, B4, C4, h_theta, tau, dt, num_time_steps)
27 error4 = compute_decoding_error(r4, C4, sigma, phi, theta,
    num_trials=500)
28
29 # --- Plot results ---
30 plt.figure(figsize=(10, 6))
31 plt.plot(time_axis, error1, label='Model 1: No Recurrence')
32 plt.plot(time_axis, error2, label='Model 2: Random Symmetric')
33 plt.plot(time_axis, error3, label='Model 3: Symmetric Ring')
34 plt.plot(time_axis, error4, label='Model 4: Balanced Ring')
35 plt.axhline(np.pi / 2, color='gray', linestyle=':', label='Chance
    level (  $\pi/2$  )')
36 plt.xlabel("Time (s)")
37 plt.ylabel("Average Decoding Error (rad)")
38 plt.title("Decoding Error Over Time (500 Trials)")
39 plt.legend()

```

```

40 plt.grid(True)
41 plt.tight_layout()
42 plt.show()

```

12.2 Simulation Prompt

Prompt: Here is my code for the simulation and decoding... I am SURE there is an error in my decoding function as the simulation's all look fine but I am getting strange values for my decoding. Can you suggest 3 ways in which I am able to debug this code to properly see what is at fault?

Model Used: ChatGPT, 4o [09/04/2025]

Output:

```

1 ... (used Cursor)

```

13 Notebooks

13.1 Python Notebook - Original

```

1 # %% [markdown]
2 # # Assignment 2 - Recurrent Neural Network Dynamics
3
4 # %% [markdown]
5 # This coursework is a study of how well a recurrent network model
6   of primary visual cortex (V1) can represent specific features
7   (we will use the example of orientation) of a brief visual
8   stimulus, in the face of readout noise, depending on its
9   connectivity.
10
11 # %% [markdown]
12 # We will begin with the experimental set up:
13 #
14 # ## Technical Set Up
15
16 # %%
17 import numpy as np
18 import matplotlib.pyplot as plt
19 from mpl_toolkits.mplot3d import Axes3D
20 from scipy.optimize import curve_fit

```

```

18
19
20 # %% [markdown]
21 # ## Experimental Set Up
22
23 # %% [markdown]
24 # ### Default Parameters
25
26 # %%
27 tau = 20 # characteristic neuron time constant (in ms)
28
29 m = 200 # number of regularly spaced orientations
30
31 n = 200 # number of neuronal inputs
32
33 B = np.eye(m,m) # feedforward input weights
34
35 C = np.eye(m,m) # matrix of output weights
36
37 sigma = 1 # noise of readout activity
38
39 k = np.pi/4 # encoder constant
40
41 alpha = 0.9 # recurrent connection strength constant 1
42
43 alpha_prime = 0.9 # recurrent connection strength constant 2
44
45 phi = 2 * np.pi * np.linspace(0,1,m) # grid of regularly spaced
    parameters
46
47 dt = 1 # timestep (in ms)
48
49 # %% [markdown]
50 # ### Default Functions
51
52 # %%
53 # Tuning Function
54 def V(z):
55     return np.exp( (np.cos(z)-1) * k**(-2))
56
57

```

```

58 # Input Encoding Function
59 def h(phi, theta):
60     return V(phi-theta)
61
62
63 # Euler Method Update Function - this is an el classico
64 def update_r(r_t, t, W, B, h_theta, tau, dt):
65     # Dirac delta pulse at t=0
66     input_term = B @ h_theta if t == 0 else 0
67
68     # Correct update equation
69     dr_dt = (1/tau) * (-r_t + W @ r_t + input_term)
70     r = r_t + dt * dr_dt
71     return r
72
73
74 # Noisy Readout of V1 Activity
75 def o_tilde_func(C, r_t, sigma):
76     noise = np.random.normal(0,1, size = C.shape[0])
77     return C @ r_t + sigma * noise
78
79
80 # Noisy Readout data
81 def theta_hat_func(phi, o_tilde):
82
83     x = np.sum(o_tilde * np.sin(phi))
84     y = np.sum(o_tilde * np.cos(phi))
85
86     # Return decoded orientation in [0, 2 )
87     return np.arctan2(x,y)
88
89
90 def circ_dist(theta_hat, theta):
91     # Circular distance in radians [0, ]
92     return np.arccos(np.cos(theta_hat - theta))
93
94
95 # Rescale matrix W_raw to the spectral radius equals alpha
96 def R(W_raw, alpha):
97     eigvals = np.linalg.eigvals(W_raw)
98     spectral_radius = np.max(np.abs(eigvals))

```

```

99     return (alpha / spectral_radius) * W_raw
100
101 # %% [markdown]
102 # ## Simulation Function
103
104 # %%
105 def run_simulation_rand(theta, phi, time_steps, W, B, C, tau, dt,
106     random_seed):
107
108     np.random.seed(random_seed)
109
110     h_model = h(phi, theta)
111     r_list = [np.zeros(W.shape[0])]
112
113     for i, time in enumerate(time_steps[:-1]):
114
115         r = update_r(r_list[i], time, W, B, h_model, tau, dt)
116
117         r_list.append(r)
118
119     return r_list[1:] # Remove initial zero state
120
121 # %% [markdown]
122 # ## Decoding Function
123
124 # %%
125 def decoding_error_func(r_list, C, sigma, phi, theta):
126
127     decoding_error_list = []
128
129     for r in r_list:
130
131         noise = np.random.normal(0, 1, size=C.shape[0])
132         o_tilde = C @ r + sigma * noise
133
134         numerator = np.sum(o_tilde * np.sin(phi))
135         denominator = np.sum(o_tilde * np.cos(phi))
136
137         theta_hat = np.arctan2(numerator, denominator)
138         decoding_error = np.arccos(np.cos(theta_hat - theta))

```



```

139         decoding_error_list.append(decoding_error)
140
141     return decoding_error_list
142
143 # %% [markdown]
144 # ## Questions and Experiments
145
146 # %% [markdown]
147 # ### Question 1 - Integrating the Dynamics of the 4 different
    Models
148
149 # %% [markdown]
150 # #### Model 1: No Recurrence
151
152 # %%
153 m = 200
154 n = 200
155 time_steps = np.linspace(0,60,100)
156 theta = np.pi
157 W_1 = np.zeros((m,n))
158 B = np.eye(m,m) # feedforward input weights
159 C = np.eye(m,m) # matrix of output weights
160
161 # Run the simulation
162 r_list_w1 = run_simulation(
163     theta = np.pi,
164     phi = 2 * np.pi * np.linspace(0,1,m),
165     time_steps = np.linspace(0,60,100),
166     W = W_1,
167     B = np.eye(m),
168     C = np.eye(m),
169     tau = 20,
170     dt = 1,
171 )
172
173 # Find the decoding error
174 decoding_error_w1 = decoding_error_func(
175     r_list = r_list_w1,
176     C = np.eye(m),
177     sigma = 1,
178     phi = 2 * np.pi * np.linspace(0,1,m),

```

```

179     theta = np.pi)
180
181
182 # %%
183 r_mat = np.array(r_list_w1) # shape: (timesteps, neurons)
184
185 plt.figure(figsize=(10, 5))
186 plt.imshow(r_mat.T, aspect='auto', cmap='viridis', origin='lower',
187            extent=[time_steps[0], time_steps[-1], 0, r_mat.shape
188                    [1]])
189 plt.colorbar(label='Firing Rate')
190 plt.xlabel('Time')
191 plt.ylabel('Neuron Index')
192 plt.title('V1 Neuron Activity Over Time (No Recurrence)')
193 plt.tight_layout()
194 plt.show()
195
196 # %%
197 # Plot
198 plt.figure(figsize=(8, 4))
199 plt.plot(time_steps, decoding_error_w1, label='Decoding Error',
200          color='purple')
201 plt.axhline(y=0, color='gray', linestyle='--', linewidth=0.8)
202 plt.xlabel('Time (ms)')
203 plt.ylabel('Decoding Error (radians)')
204 plt.title('Decoding Error Over Time (No Recurrence)')
205 plt.grid(True)
206 plt.legend()
207 plt.tight_layout()
208 plt.show()
209
210 # %%
211 # Convert time steps to indices
212 idx_0p = 0
213 idx_20 = np.argmin(np.abs(time_steps - 20))
214 idx_60 = np.argmin(np.abs(time_steps - 60))
215
216 # Extract r(t) vectors
217 r_0p = r_list_w1[idx_0p]
218 r_20 = r_list_w1[idx_20]

```

```

218 r_60 = r_list_w1[idx_60]
219
220 phi = 2 * np.pi * np.linspace(0, 1, m) # match your code
221
222 plt.figure(figsize=(10, 4))
223 plt.plot(phi, r_0p, label='t = 0 ms')
224 plt.plot(phi, r_20, label='t = 20 ms')
225 plt.plot(phi, r_60, label='t = 60 ms')
226 plt.xlabel('Preferred Orientation (radians)')
227 plt.ylabel('Firing rate r(t)')
228 plt.title('V1 Population Response Over Time (No Recurrence)')
229 plt.legend()
230 plt.grid(True)
231 plt.tight_layout()
232 plt.show()
233
234
235 # %% [markdown]
236 # #### Model 2: Random Symmetric Connectivity
237
238 # %%
239 # Model 2: Random symmetric connectivity
240 B = np.eye(m,m) # feedforward input weights
241 C = np.eye(m,m) # matrix of output weights
242 W_tilde = np.random.normal(0, 1, size=B.shape)
243 W_2 = R(W_tilde + W_tilde.T, alpha)
244
245 m = 200
246 n = 200
247 time_steps = np.linspace(0,60,100)
248 theta = np.pi
249
250 # Run the simulation
251 r_list_w2 = run_simulation(
252     theta = np.pi,
253     phi = 2 * np.pi * np.linspace(0,1,m),
254     time_steps = np.linspace(0,60,100),
255     W = W_2,
256     B = np.eye(m),
257     C = np.eye(m),
258     tau = 20,

```

```

259     dt = 1,
260 )
261
262 # Find the decoding error
263 decoding_error_w2 = decoding_error_func(
264     r_list = r_list_w2,
265     C = np.eye(m),
266     sigma = 1,
267     phi = 2 * np.pi * np.linspace(0,1,m),
268     theta = np.pi)
269
270 # %% [markdown]
271 # ##### Plotting
272
273 # %%
274 r_mat_w2 = np.array(r_list_w2) # shape: (timesteps, neurons)
275
276 plt.figure(figsize=(10, 5))
277 plt.imshow(r_mat_w2.T, aspect='auto', cmap='viridis', origin='
    lower',
278             extent=[time_steps[0], time_steps[-1], 0, r_mat.shape
                [1]])
279 plt.colorbar(label='Firing Rate')
280 plt.xlabel('Time')
281 plt.ylabel('Neuron Index')
282 plt.title('V1 Neuron Activity Over Time (Random Symmetric
    Connectivity)')
283 plt.tight_layout()
284 plt.show()
285
286
287 # %%
288 # Plot
289 plt.figure(figsize=(8, 4))
290 plt.plot(time_steps, decoding_error_w2, label='Decoding Error',
    color='purple')
291 plt.axhline(y=0, color='gray', linestyle='--', linewidth=0.8)
292 plt.xlabel('Time (ms)')
293 plt.ylabel('Decoding Error (radians)')
294 plt.title('Decoding Error Over Time ((Random Symmetric
    Connectivity)')

```

```

295 plt.grid(True)
296 plt.legend()
297 plt.tight_layout()
298 plt.show()
299
300 # %%
301 # Convert time steps to indices
302 idx_0p = 0
303 idx_20 = np.argmin(np.abs(time_steps - 20))
304 idx_60 = np.argmin(np.abs(time_steps - 60))
305
306 # Extract r(t) vectors
307 r_0p = r_list_w2[idx_0p]
308 r_20 = r_list_w2[idx_20]
309 r_60 = r_list_w2[idx_60]
310
311 phi = 2 * np.pi * np.linspace(0, 1, m) # match your code
312
313 plt.figure(figsize=(10, 4))
314 plt.plot(phi, r_0p, label='t = 0 ms')
315 plt.plot(phi, r_20, label='t = 20 ms')
316 plt.plot(phi, r_60, label='t = 60 ms')
317 plt.xlabel('Preferred Orientation (radians)')
318 plt.ylabel('Firing rate r(t)')
319 plt.title('V1 Population Response Over Time (Random Symmetric
    Connectivity)')
320 plt.legend()
321 plt.grid(True)
322 plt.tight_layout()
323 plt.show()
324
325
326 # %%
327 print(np.linalg.norm(np.array(r_list_w1) - np.array(r_list_w2)))
328 print(np.linalg.norm(np.array(decoding_error_w1) - np.array(
    decoding_error_w2)))
329
330
331 # %% [markdown]
332 # So the above shows that these in fact ARE changing - but just
    not very much! Hence, the recurrence isn't strong or

```

```

directional enough. The stimulus is brief and noise masks the
effect!
333
334 # %% [markdown]
335 # ##### Model 3: Symmetric Ring Structure
336
337 # %%
338 W_ij = np.zeros_like(B)
339 for i in range(n):
340     for j in range(n):
341         W_ij[i,j] = V(phi[i]-phi[j])
342
343 W_3 = R(W_ij, alpha)
344
345
346 # Run the simulation
347 r_list_w3 = run_simulation(
348     theta = np.pi,
349     phi = 2 * np.pi * np.linspace(0,1,m),
350     time_steps = np.linspace(0,60,100),
351     W = W_3,
352     B = np.eye(m),
353     C = np.eye(m),
354     tau = 20,
355     dt = 1,
356 )
357
358 # Find the decoding error
359 decoding_error_w3 = decoding_error_func(
360     r_list = r_list_w3,
361     C = np.eye(m),
362     sigma = 1,
363     phi = 2 * np.pi * np.linspace(0,1,m),
364     theta = np.pi)
365
366
367 # %%
368 r_mat_w3 = np.array(r_list_w3) # shape: (timesteps, neurons)
369
370 plt.figure(figsize=(10, 5))

```

```

371 plt.imshow(r_mat_w3.T, aspect='auto', cmap='viridis', origin='
    lower',
372             extent=[time_steps[0], time_steps[-1], 0, r_mat_w3.
                shape[1]])
373 plt.colorbar(label='Firing Rate')
374 plt.xlabel('Time')
375 plt.ylabel('Neuron Index')
376 plt.title('V1 Neuron Activity Over Time (Symmetric Ring Structure)
    ')
377 plt.tight_layout()
378 plt.show()
379
380
381 # %%
382 # Plot
383 plt.figure(figsize=(8, 4))
384 plt.plot(time_steps, decoding_error_w3, label='Decoding Error',
    color='purple')
385 plt.axhline(y=0, color='gray', linestyle='--', linewidth=0.8)
386 plt.xlabel('Time (ms)')
387 plt.ylabel('Decoding Error (radians)')
388 plt.title('Decoding Error Over Time ((Random Symmetric
    Connectivity)')
389 plt.grid(True)
390 plt.legend()
391 plt.tight_layout()
392 plt.show()
393
394 # %%
395 # Convert time steps to indices
396 idx_0p = 0
397 idx_20 = np.argmin(np.abs(time_steps - 20))
398 idx_60 = np.argmin(np.abs(time_steps - 60))
399
400 # Extract r(t) vectors
401 r_0p = r_list_w3[idx_0p]
402 r_20 = r_list_w3[idx_20]
403 r_60 = r_list_w3[idx_60]
404
405 phi = 2 * np.pi * np.linspace(0, 1, m) # match your code
406

```

```

407 plt.figure(figsize=(10, 4))
408 plt.plot(phi, r_0p, label='t = 0 ms')
409 plt.plot(phi, r_20, label='t = 20 ms')
410 plt.plot(phi, r_60, label='t = 60 ms')
411 plt.xlabel('Preferred Orientation (radians)')
412 plt.ylabel('Firing rate r(t)')
413 plt.title('V1 Population Response Over Time (Random Symmetric
Connectivity)')
414 plt.legend()
415 plt.grid(True)
416 plt.tight_layout()
417 plt.show()
418
419
420 # %% [markdown]
421 # #### Model 4: Balanced Ring Structure
422
423 # %%
424 # Model 4: Balanced Ring Structure
425
426 m = 200
427 n = 2*m
428
429 W_4 = np.block([
430     [ W_3, -W_3],
431     [ W_3, -W_3]
432 ]) # (2m, 2m)
433
434 B_4 = np.vstack([np.eye(m), np.zeros((m, m))]) # (2m, m)
435 C_4 = np.hstack([np.eye(m), np.zeros((m, m))]) # (m, 2m)
436
437 time_steps = np.linspace(0,60,100)
438 theta = np.pi
439
440 # Run the simulation
441 r_list_w4 = run_simulation(
442     theta = np.pi,
443     phi = 2 * np.pi * np.linspace(0,1,m),
444     time_steps = np.linspace(0,60,100),
445     W = W_4,
446     B = B_4,

```



```

447     C = C_4,
448     tau = 20,
449     dt = 1,
450 )
451
452 # Find the decoding error
453 decoding_error_w4 = decoding_error_func(
454     r_list = r_list_w4,
455     C = C_4,
456     sigma = 1,
457     phi = 2 * np.pi * np.linspace(0,1,m),
458     theta = np.pi)
459
460 # %% [markdown]
461 # #### Plotting
462
463 # %%
464 r_mat_w4 = np.array(r_list_w4) # shape: (timesteps, 2m)
465
466 plt.figure(figsize=(10, 5))
467 plt.imshow(r_mat_w4.T, aspect='auto', cmap='viridis', origin='
    lower',
468             extent=[time_steps[0], time_steps[-1], 0, r_mat_w4.
                shape[1]])
469 plt.colorbar(label='Firing Rate')
470 plt.xlabel('Time')
471 plt.ylabel('Neuron Index')
472 plt.title('V1 Neuron Activity Over Time (Balanced Ring Structure)'
    )
473 plt.tight_layout()
474 plt.show()
475
476
477 # %%
478 plt.figure(figsize=(8, 4))
479 plt.plot(time_steps, decoding_error_w4, label='Decoding Error',
    color='purple')
480 plt.axhline(y=0, color='gray', linestyle='--', linewidth=0.8)
481 plt.xlabel('Time (ms)')
482 plt.ylabel('Decoding Error (radians)')
483 plt.title('Decoding Error Over Time (Balanced Ring Structure)')

```

```

484 plt.grid(True)
485 plt.legend()
486 plt.tight_layout()
487 plt.show()
488
489
490 # %%
491 idx_0p = 0
492 idx_20 = np.argmin(np.abs(time_steps - 20))
493 idx_60 = np.argmin(np.abs(time_steps - 60))
494
495 # Extract first m neurons only (since C reads from them)
496 r_0p = r_list_w4[idx_0p][:m]
497 r_20 = r_list_w4[idx_20][:m]
498 r_60 = r_list_w4[idx_60][:m]
499
500 phi = 2 * np.pi * np.linspace(0, 1, m)
501
502 plt.figure(figsize=(10, 4))
503 plt.plot(phi, r_0p, label='t = 0      ms')
504 plt.plot(phi, r_20, label='t = 20 ms')
505 plt.plot(phi, r_60, label='t = 60 ms')
506 plt.xlabel('Preferred Orientation      (radians)')
507 plt.ylabel('Firing rate r(t)')
508 plt.title('V1 Population Response Over Time (Balanced Ring
          Structure)')
509 plt.legend()
510 plt.grid(True)
511 plt.tight_layout()
512 plt.show()
513
514
515 # %% [markdown]
516 # #### Report Graphic
517
518 # %%
519 # Indices for t = 0      , 20 ms, 60 ms
520 idx_0p = 0
521 idx_20 = np.argmin(np.abs(time_steps - 20))
522 idx_60 = np.argmin(np.abs(time_steps - 60))
523

```

```

524 phi = 2 * np.pi * np.linspace(0, 1, m)
525
526 # Helper function to extract and plot
527 def plot_tuning(ax, r_list, label):
528     r_0p = r_list[idx_0p][:m]
529     r_20 = r_list[idx_20][:m]
530     r_60 = r_list[idx_60][:m]
531
532     ax.plot(phi, r_0p, label='t = 0    ms')
533     ax.plot(phi, r_20, label='t = 20 ms')
534     ax.plot(phi, r_60, label='t = 60 ms')
535     ax.set_title(label)
536     ax.set_xlabel('Preferred Orientation    (rad)')
537     ax.set_ylabel('Firing rate r(t)')
538     ax.grid(True)
539
540 # Plot all four
541 fig, axs = plt.subplots(2, 2, figsize=(14, 10))
542 plot_tuning(axs[0, 0], r_list_w1, 'Model 1: No Recurrence')
543 plot_tuning(axs[0, 1], r_list_w2, 'Model 2: Random Symmetric')
544 plot_tuning(axs[1, 0], r_list_w3, 'Model 3: Ring Structure')
545 plot_tuning(axs[1, 1], r_list_w4, 'Model 4: Balanced Ring')
546
547 # Tidy up
548 for ax in axs.flatten():
549     ax.legend()
550
551 plt.suptitle('V1 Population Tuning Curves at t = 0    , 20 ms, 60
552             ms', fontsize=16)
553 plt.tight_layout(rect=[0, 0, 1, 0.97])
554 plt.show()
555
556 # %% [markdown]
557 # ### Question 2: Investigations into Model 2
558
559 # %% [markdown]
560 # We begin by investigating the variance against time for all 4
561     fitted models, presuming that at each discretised time step we
562     can fit a Gaussian. We then measure the "noise" as the
563     difference between the actual activity and the fitted Gaussian:

```

the residual. We then compute the standard deviation of the residuals for each model over time - thus capturing irregularities and small-scale noise.

```

561
562 # %%
563 import numpy as np
564 import matplotlib.pyplot as plt
565 from scipy.stats import norm
566
567 # Get real parts of eigenvalues
568 eigvals_w2 = np.linalg.eigvals(W_2).real
569 eigvals_w3 = np.linalg.eigvals(W_3).real
570
571 # Create x-axis range for KDE
572 x = np.linspace(-2, 2, 1000)
573
574 # KDE function
575 def kde_curve(eigvals, bandwidth=0.05):
576     density = np.zeros_like(x)
577     for val in eigvals:
578         density += norm.pdf(x, loc=val, scale=bandwidth)
579     return density / len(eigvals)
580
581 # Compute KDEs
582 density_w2 = kde_curve(eigvals_w2)
583 density_w3 = kde_curve(eigvals_w3)
584
585 # Plot
586 plt.figure(figsize=(10, 4))
587 plt.plot(x, density_w2, label='Model 2 (Random Symmetric)',
588         linewidth=2)
589 plt.plot(x, density_w3, label='Model 3 (Ring Structure)',
590         linestyle='--', linewidth=2)
591
592 plt.axvline(1, color='red', linestyle=':', label=' = 1 (
593     Instability)')
594 plt.xlabel('Eigenvalue (Real Part)')
595 plt.ylabel('Density')
596 plt.title('Eigenvalue Spectrum Comparison:  $W^{(2)}$  vs  $W^{(3)}$ 
597     ')
598 plt.legend()

```

```

595 plt.grid(True)
596 plt.tight_layout()
597 plt.show()
598
599
600 # %% [markdown]
601 # ### Question 3: Investigation into Model 3
602
603 # %%
604 def total_response(r_list):
605     return [np.linalg.norm(r)**2 for r in r_list]
606
607 resp_2 = total_response(r_list_w2)
608 resp_3 = total_response(r_list_w3)
609
610 plt.plot(time_steps, resp_2, label='Model 2')
611 plt.plot(time_steps, resp_3, label='Model 3')
612 plt.xlabel("Time (ms)")
613 plt.ylabel("Total Population Activity ( r (t) )")
614 plt.title("Model 3 Shows Stronger Population Response")
615 plt.legend()
616 plt.grid(True)
617 plt.tight_layout()
618 plt.show()
619
620
621 # %% [markdown]
622 # # Question 4
623 #
624
625 # %%
626 # Euler Method Update Function - this is an el classico
627 def update_r(r_t, t, W, B, h_theta, tau, dt):
628     # Dirac delta pulse at t=0
629     input_term = B @ h_theta if t == 0 else 0
630
631     # Correct update equation
632     dr_dt = (1/tau) * (-r_t + W @ r_t + input_term)
633     r = r_t + dt * dr_dt
634     return r
635

```

```

636 # %%
637 import random
638
639 def run_simulation_rand(theta, phi, time_steps, W, B, C, tau, dt,
640     random_seed):
641
642     random.seed(random_seed)
643
644     # --- Main simulation ---
645
646     # Vector: V(phi-theta), where phi is the grid of regularly
647     # spaced points, and theta is the input orientation
648     # The V function is a Von-Mises distribution, commonly used to
649     # model data measured on a circle (e.g. orientation)
650     h_model = h(phi, theta)
651
652     # Vector: the ith element of this vector represents the
653     # momentary firing rate of V1 neuron i
654     r_list = [np.zeros(W.shape[0])] # added a dummy value at the
655     # start so by code doesn't f*ck up
656
657     # for each time step
658     for i, time in enumerate(time_steps):
659
660         # r = update r given the previous one
661         r = update_r(r_list[i], time, W, B, h_model, tau, dt)
662
663         r_list.append(r) # appending r to my list
664
665     r_list = r_list[1:] # getting rid of dummy value - is this
666     # optimal coding
667
668     return r_list
669
670 # %%
671 def decoding_error_func(r_list, C, sigma, phi, theta):
672
673     decoding_error_list = []
674
675     for r in r_list: # r is a vector size n!

```

```

671     noise = np.random.normal(0,1, size = C.shape[0])
672
673
674     o_tilde = (C @ r) + (sigma * noise)
675
676     x = np.sum(o_tilde * np.sin(phi))
677     y = np.sum(o_tilde * np.cos(phi))
678
679     theta_hat = np.arctan2(x,y)
680
681     decoding_error = np.arccos(np.cos(theta_hat-theta))
682
683     decoding_error_list.append(decoding_error)
684
685     return decoding_error_list
686
687
688 # %%
689 tau = 0.02 # characteristic neuron time constant (in m)
690
691 m = 200 # number of regularly spaced orientations
692
693 n = 200 # number of neuronal inputs
694
695 B = np.eye(m,m) # feedforward input weights
696
697 C = np.eye(m,m) # matrix of output weights
698
699 sigma = 1 # noise of readout activity
700
701 k = np.pi/4 # encoder constant
702
703 alpha = 0.9 # recurrent connection strength constant 1
704
705 alpha_prime = 0.9 # recurrent connection strength constant 2
706
707 phi = 2 * np.pi * np.linspace(0,1,m) # grid of regularly spaced
708     parameters
709
710 dt = 0.001 # timestep (in s)

```

```

711
712 # %% [markdown]
713 # ##### Model 1
714
715 # %%
716 theta = np.pi
717 n_trials = 100
718 time_steps = np.linspace(0, 0.06, num=60)
719 W_1 = np.zeros((m,m))
720
721 decoding_errors_w1 = []
722
723 for trial in range(n_trials):
724
725     r_list = run_simulation_rand(theta, phi, time_steps, W_1, B, C
726     , tau, dt, random_seed=trial)
727     err = decoding_error_func(r_list, C, sigma, phi, theta)
728     decoding_errors_w1.append(err)
729
730 # Average over trials
731 avg_error_w1 = np.mean(decoding_errors_w1, axis=0)
732
733
734 # %% [markdown]
735 # ##### Model 2
736
737 # %%
738 W_tilde = np.random.normal(0, 1, size=B.shape)
739 W_2 = R(W_tilde + W_tilde.T, alpha)
740
741 # %%
742 decoding_errors_w2 = []
743 for trial in range(n_trials):
744     r_list = run_simulation_rand(theta, phi, time_steps, W_2, B, C
745     , tau, dt, random_seed=trial)
746     err = decoding_error_func(r_list, C, sigma, phi, theta)
747     decoding_errors_w2.append(err)
748
749 # Average over trials
750 avg_error_w2 = np.mean(decoding_errors_w2, axis=0)

```



```

750
751 # %% [markdown]
752 # ##### Model 3
753
754 # %%
755 W_ij = np.zeros_like(B)
756 for i in range(m):
757     for j in range(m):
758         W_ij[i,j] = V(phi[i]-phi[j])
759
760 W_3 = R(W_ij, alpha)
761
762 decoding_errors_w3 = []
763 for trial in range(n_trials):
764     r_list = run_simulation_rand(theta, phi, time_steps, W_3, B, C
765         , tau, dt, random_seed=trial)
766     err = decoding_error_func(r_list, C, sigma, phi, theta)
767     decoding_errors_w3.append(err)
768
769 # Average over trials
770 avg_error_w3 = np.mean(decoding_errors_w3, axis=0)
771
772 # %% [markdown]
773 # ##### Model 4
774
775 # %%
776 m = 200
777 n = 2*m
778
779 W_4 = np.block([
780     [ W_3, -W_3],
781     [ W_3, -W_3]
782 ]) # (2m, 2m)
783
784 B_4 = np.vstack([np.eye(m), np.zeros((m, m))]) # (2m, m)
785 C_4 = np.hstack([np.eye(m), np.zeros((m, m))]) # (m, 2m)
786
787 decoding_errors_w4 = []
788 for trial in range(n_trials):
789     r_list = run_simulation_rand(theta, phi, time_steps, W_4, B_4,
790         C_4, tau, dt, random_seed=trial)

```

```

789     err = decoding_error_func(r_list, C_4, sigma, phi, theta)
790     decoding_errors_w4.append(err)
791
792 # Average over trials
793 avg_error_w4 = np.mean(decoding_errors_w4, axis = 0)
794
795 # %%
796 plt.figure(figsize=(10, 5))
797 plt.plot(time_steps, avg_error_w1, label='Model 1')
798 plt.plot(time_steps, avg_error_w2, label='Model 2')
799 plt.plot(time_steps, avg_error_w3, label='Model 3')
800 plt.plot(time_steps, avg_error_w4, label='Model 4')
801 plt.xlabel("Time (ms)")
802 plt.ylabel("Mean Decoding Error (radians)")
803 plt.title("Average Decoding Error Over Time (    =    )")
804 plt.legend()
805 plt.grid(True)
806 plt.tight_layout()
807 plt.show()
808
809
810 # %%
811 import numpy as np
812 from scipy import linalg
813
814 # --- Parameters ---
815 m = 200
816 phi = np.linspace(0, 2 * np.pi, m, endpoint=False)
817 alpha = 0.9
818 alpha_prime = 0.9
819 kappa = np.pi / 4
820
821 # --- Tuning function ---
822 def V(z, kappa=kappa):
823     return np.exp((np.cos(z) - 1) / (kappa**2))
824
825 # --- Rescale W to have spectral abscissa ---
826 def rescale_matrix(W, target_alpha):
827     if np.all(W == 0):
828         return W
829     eigvals = linalg.eigvals(W)

```

```

830     max_real = np.max(np.real(eigvals))
831     return W * (target_alpha / max_real) if max_real != 0 else W
832
833
834 # %%
835 W1 = np.zeros((m, m))
836
837 W_tilde = np.random.normal(0, 1, (m, m))
838 W2 = rescale_matrix(W_tilde + W_tilde.T, alpha)
839
840 W3 = np.zeros((m, m))
841 for i in range(m):
842     for j in range(m):
843         W3[i, j] = V(phi[i] - phi[j])
844 W3 = rescale_matrix(W3, alpha)
845
846 W_ring = np.zeros((m, m))
847 for i in range(m):
848     for j in range(m):
849         W_ring[i, j] = V(phi[i] - phi[j])
850 W_ring = rescale_matrix(W_ring, alpha_prime)
851
852 n_model = 2 * m # Double the neurons
853
854 W4 = np.zeros((n_model, n_model))
855 W4[:m, :m] = W_ring # E E
856 W4[:m, m:] = -W_ring # I E (inhibitory input)
857 W4[m:, :m] = W_ring # E I
858 W4[m:, m:] = -W_ring # I I (self-inhibition)
859
860
861
862
863
864 # %%
865 import numpy as np
866 import matplotlib.pyplot as plt
867 from scipy import linalg
868
869 # --- Parameters ---
870 m = 200

```

```

871 tau = 0.02
872 dt = 0.001
873 kappa = np.pi / 4
874 alpha = 0.9
875 sigma = 1
876 theta = np.pi
877 duration = 0.06
878 num_time_steps = int(duration / dt) + 1
879 phi = np.linspace(0, 2 * np.pi, m, endpoint=False)
880
881 # --- Von Mises tuning + stimulus encoding ---
882 def stimulus_encoding(phi_i, theta, kappa=kappa):
883     return np.exp((np.cos(phi_i - theta) - 1) / (kappa**2))
884
885 def h_theta_vector(phi, theta):
886     return stimulus_encoding(phi, theta)
887
888 # --- Rescale matrix by spectral abscissa ---
889 def rescale_matrix(W, target_alpha):
890     if np.all(W == 0):
891         return W
892     eigvals = linalg.eigvals(W)
893     max_real = np.max(np.real(eigvals))
894     return W * (target_alpha / max_real) if max_real != 0 else W
895
896 def network_model(model_num, m, phi, alpha, alpha_prime):
897     def V(z, kappa=np.pi/4):
898         return np.exp((np.cos(z) - 1) / kappa**2)
899
900     def rescale_matrix(W, target_alpha):
901         if np.all(W == 0):
902             return W
903         eigvals = linalg.eigvals(W)
904         max_real = np.max(np.real(eigvals))
905         return W * (target_alpha / max_real) if max_real != 0 else
906             W
907
908     if model_num == 1:
909         # Model 1: No recurrence
910         W = np.zeros((m, m))
911         B = np.eye(m)

```

```

911     C = np.eye(m)
912     n_model = m
913
914     elif model_num == 2:
915         # Model 2: Random symmetric
916         W_tilde = np.random.normal(0, 1, (m, m))
917         W = rescale_matrix(W_tilde + W_tilde.T, alpha)
918         B = np.eye(m)
919         C = np.eye(m)
920         n_model = m
921
922     elif model_num == 3:
923         # Model 3: Symmetric ring structure
924         W = np.zeros((m, m))
925         for i in range(m):
926             for j in range(m):
927                 W[i, j] = V(phi[i] - phi[j])
928         W = rescale_matrix(W, alpha)
929         B = np.eye(m)
930         C = np.eye(m)
931         n_model = m
932
933     elif model_num == 4:
934         # Model 4: Balanced ring structure (E/I split)
935         W_ring = np.zeros((m, m))
936         for i in range(m):
937             for j in range(m):
938                 W_ring[i, j] = V(phi[i] - phi[j])
939         W_ring = rescale_matrix(W_ring, alpha_prime)
940
941         n_model = 2 * m
942         W = np.zeros((n_model, n_model))
943         W[:m, :m] = W_ring          # E E
944         W[:m, m:] = -W_ring         # I E
945         W[m:, :m] = W_ring          # E I
946         W[m:, m:] = -W_ring         # I I
947
948         B = np.zeros((n_model, m))
949         B[:m, :] = np.eye(m)
950
951         C = np.zeros((m, n_model))

```

```

952         C[:, :m] = np.eye(m)
953
954     else:
955         raise ValueError(f"Model {model_num} is not implemented (
956             must be 1 4 ).")
957
958     return W, B, C, n_model
959
960 # --- Simulation ---
961 def simulate_model(W, B, C, h_theta, tau, dt, num_time_steps):
962     n_model = W.shape[0]
963     r = np.zeros((n_model, num_time_steps))
964     r[:, 0] = B @ h_theta
965     for i in range(1, num_time_steps):
966         dr = (-r[:, i-1] + W @ r[:, i-1]) * (dt / tau)
967         r[:, i] = r[:, i-1] + dr
968     return r
969
970 # --- Decode (t) and compute circular distance error ---
971 def compute_decoding_error(r, C, sigma, phi, theta, num_trials
972     =100):
973     m, T = C.shape[0], r.shape[1]
974     all_errors = []
975     for _ in range(num_trials):
976         trial_errors = []
977         for t in range(T):
978             noise = sigma * np.random.randn(m)
979             o_tilde = C @ r[:, t] + noise
980             x = np.sum(o_tilde * np.cos(phi))
981             y = np.sum(o_tilde * np.sin(phi))
982             theta_hat = np.arctan2(y, x) % (2 * np.pi)
983             error = np.abs(np.angle(np.exp(1j * (theta_hat - theta
984                 ))))
985             trial_errors.append(error)
986         all_errors.append(trial_errors)
987     return np.mean(all_errors, axis=0)
988
989 # --- Run everything for Model 1 and Model 2 ---
990 time_axis = np.linspace(0, duration, num_time_steps)
991 h_theta = h_theta_vector(phi, theta)

```

```

990
991 # --- Run everything for Models 1 to 4 ---
992 time_axis = np.linspace(0, duration, num_time_steps)
993 h_theta = h_theta_vector(phi, theta)
994
995 # Model 1
996 W1, B1, C1, _ = network_model(1, m, phi, alpha, alpha_prime)
997 r1 = simulate_model(W1, B1, C1, h_theta, tau, dt, num_time_steps)
998 error1 = compute_decoding_error(r1, C1, sigma, phi, theta,
    num_trials=100)
999
1000 # Model 2
1001 W2, B2, C2, _ = network_model(2, m, phi, alpha, alpha_prime)
1002 r2 = simulate_model(W2, B2, C2, h_theta, tau, dt, num_time_steps)
1003 error2 = compute_decoding_error(r2, C2, sigma, phi, theta,
    num_trials=100)
1004
1005 # Model 3
1006 W3, B3, C3, _ = network_model(3, m, phi, alpha, alpha_prime)
1007 r3 = simulate_model(W3, B3, C3, h_theta, tau, dt, num_time_steps)
1008 error3 = compute_decoding_error(r3, C3, sigma, phi, theta,
    num_trials=100)
1009
1010 # Model 4
1011 W4, B4, C4, _ = network_model(4, m, phi, alpha, alpha_prime)
1012 r4 = simulate_model(W4, B4, C4, h_theta, tau, dt, num_time_steps)
1013 error4 = compute_decoding_error(r4, C4, sigma, phi, theta,
    num_trials=100)
1014
1015 # --- Plot results ---
1016 plt.figure(figsize=(10, 6))
1017 plt.plot(time_axis, error1, label='Model 1: No Recurrence')
1018 plt.plot(time_axis, error2, label='Model 2: Random Symmetric')
1019 plt.plot(time_axis, error3, label='Model 3: Symmetric Ring')
1020 plt.plot(time_axis, error4, label='Model 4: Balanced Ring')
1021 plt.axhline(np.pi / 2, color='gray', linestyle=':', label='Chance
    level (  $\pi/2$  )')
1022 plt.xlabel("Time (s)")
1023 plt.ylabel("Average Decoding Error (rad)")
1024 plt.title("Decoding Error Over Time (100 Trials)")
1025 plt.legend()

```

```

1026 plt.grid(True)
1027 plt.tight_layout()
1028 plt.show()

```

13.2 Python Notebook - with Changes

```

1  # %% [markdown]
2  # # Assignment 2 - Recurrent Neural Network Dynamics
3
4  # %%
5  # --- Imports ---
6  import numpy as np
7  import matplotlib.pyplot as plt
8  from scipy import linalg
9
10 # %%
11 # --- Parameters ---
12 m = 200
13 tau = 0.02
14 dt = 0.001
15 kappa = np.pi / 4
16 alpha = 0.9
17 sigma = 1
18 theta = np.pi
19 duration = 0.06
20 alpha_prime = 0.9
21 num_time_steps = int(duration / dt) + 1
22 phi = np.linspace(0, 2 * np.pi, m, endpoint=False)
23
24 # %%
25 # --- All Functions ---
26
27 # --- Von Mises tuning + stimulus encoding ---
28 def stimulus_encoding(phi_i, theta, kappa=kappa):
29     return np.exp((np.cos(phi_i - theta) - 1) / (kappa**2))
30
31 def h_theta_vector(phi, theta):
32     return stimulus_encoding(phi, theta)
33
34 # --- Rescale matrix by spectral abscissa ---
35 def rescale_matrix(W, target_alpha):

```



```

36     if np.all(W == 0):
37         return W
38     eigvals = linalg.eigvals(W)
39     max_real = np.max(np.real(eigvals))
40     return W * (target_alpha / max_real) if max_real != 0 else W
41
42 def network_model(model_num, m, phi, alpha, alpha_prime):
43     def V(z, kappa=np.pi/4):
44         return np.exp((np.cos(z) - 1) / kappa**2)
45
46     def rescale_matrix(W, target_alpha):
47         if np.all(W == 0):
48             return W
49         eigvals = linalg.eigvals(W)
50         max_real = np.max(np.real(eigvals))
51         return W * (target_alpha / max_real) if max_real != 0 else
52             W
53
54     if model_num == 1:
55         # Model 1: No recurrence
56         W = np.zeros((m, m))
57         B = np.eye(m)
58         C = np.eye(m)
59         n_model = m
60
61     elif model_num == 2:
62         # Model 2: Random symmetric
63         W_tilde = np.random.normal(0, 1, (m, m))
64         W = rescale_matrix(W_tilde + W_tilde.T, alpha)
65         B = np.eye(m)
66         C = np.eye(m)
67         n_model = m
68
69     elif model_num == 3:
70         # Model 3: Symmetric ring structure
71         W = np.zeros((m, m))
72         for i in range(m):
73             for j in range(m):
74                 W[i, j] = V(phi[i] - phi[j])
75         W = rescale_matrix(W, alpha)
76         B = np.eye(m)

```

```

76     C = np.eye(m)
77     n_model = m
78
79     elif model_num == 4:
80         # Model 4: Balanced ring structure (E/I split)
81         W_ring = np.zeros((m, m))
82         for i in range(m):
83             for j in range(m):
84                 W_ring[i, j] = V(phi[i] - phi[j])
85         W_ring = rescale_matrix(W_ring, alpha_prime)
86
87         n_model = 2 * m
88         W = np.zeros((n_model, n_model))
89         W[:m, :m] = W_ring          # E E
90         W[:m, m:] = -W_ring         # I E
91         W[m:, :m] = W_ring          # E I
92         W[m:, m:] = -W_ring         # I I
93
94         B = np.zeros((n_model, m))
95         B[:m, :] = np.eye(m)
96
97         C = np.zeros((m, n_model))
98         C[:, :m] = np.eye(m)
99
100    elif model_num == 5:
101        # Model 5: Balanced Ring with Balanced Input (push-pull)
102        W_ring = np.zeros((m, m))
103        for i in range(m):
104            for j in range(m):
105                W_ring[i, j] = V(phi[i] - phi[j])
106        W_ring = rescale_matrix(W_ring, alpha_prime)
107
108        n_model = 2 * m
109        W = np.zeros((n_model, n_model))
110        W[:m, :m] = W_ring
111        W[:m, m:] = -W_ring
112        W[m:, :m] = W_ring
113        W[m:, m:] = -W_ring
114
115        B = np.zeros((n_model, m))
116        B[:m, :] = np.eye(m)

```

```

117         B[m:, :] = -np.eye(m)
118
119         C = np.zeros((m, n_model))
120         C[:, :m] = np.eye(m)
121
122     else:
123         raise ValueError(f"Model {model_num} is not implemented (
124                             must be 1 5 ).")
125
126     return W, B, C, n_model
127
128 # --- Simulation ---
129 def simulate_model(W, B, C, h_theta, tau, dt, num_time_steps):
130     n_model = W.shape[0]
131     r = np.zeros((n_model, num_time_steps))
132     r[:, 0] = B @ h_theta
133     for i in range(1, num_time_steps):
134         dr = (-r[:, i-1] + W @ r[:, i-1]) * (dt / tau)
135         r[:, i] = r[:, i-1] + dr
136     return r
137
138 # --- Decode (t) and compute circular distance error ---
139 def compute_decoding_error(r, C, sigma, phi, theta, num_trials
140                             =100):
141     m, T = C.shape[0], r.shape[1]
142     all_errors = []
143     for _ in range(num_trials):
144         trial_errors = []
145         for t in range(T):
146             noise = sigma * np.random.randn(m)
147             o_tilde = C @ r[:, t] + noise
148             x = np.sum(o_tilde * np.cos(phi))
149             y = np.sum(o_tilde * np.sin(phi))
150             theta_hat = np.arctan2(y, x) % (2 * np.pi)
151             error = np.abs(np.angle(np.exp(1j * (theta_hat - theta
152                                     ))))
153             trial_errors.append(error)
154         all_errors.append(trial_errors)
155     return np.mean(all_errors, axis=0)

```

```

155 # %% [markdown]
156 # ## Question 1
157
158 # %%
159 import matplotlib.pyplot as plt
160
161 # --- Simulation settings ---
162 m = 200
163 tau = 0.02
164 dt = 0.001
165 duration = 0.06 # 60 ms
166 num_time_steps = int(duration / dt) + 1
167 phi = np.linspace(0, 2 * np.pi, m, endpoint=False)
168 theta = np.pi
169 alpha = 0.9
170 alpha_prime = 0.9
171 time_axis = np.linspace(0, duration, num_time_steps)
172
173 # Times of interest in indices
174 t_0p = 0
175 t_20 = int(0.020 / dt)
176 t_60 = int(0.060 / dt)
177
178 # Create figure
179 fig, axs = plt.subplots(2, 2, figsize=(12, 8))
180 titles = [
181     "Model 1: No Recurrence",
182     "Model 2: Random Symmetric",
183     "Model 3: Symmetric Ring",
184     "Model 4: Balanced Ring"
185 ]
186
187 # Loop through all 4 models
188 for model_num in range(1, 5):
189     # Setup model
190     W, B, C, n_model = network_model(model_num, m, phi, alpha,
191                                     alpha_prime)
192     h_theta = h_theta_vector(phi, theta)
193     r = simulate_model(W, B, C, h_theta, tau, dt, num_time_steps)
194
195     # Extract r(t) at 0 , 20ms, 60ms

```

```

195     r_0p = r[:,m, t_0p]    # always plot only the first m neurons (
        excitatory)
196     r_20 = r[:,m, t_20]
197     r_60 = r[:,m, t_60]
198
199     # Subplot position
200     ax = axs[(model_num - 1) // 2][(model_num - 1) % 2]
201
202     ax.plot(phi, r_0p, label='t = 0    ms')
203     ax.plot(phi, r_20, label='t = 20 ms')
204     ax.plot(phi, r_60, label='t = 60 ms')
205     ax.set_title(titles[model_num - 1])
206     ax.set_xlabel('Preferred Orientation    (radians)')
207     ax.set_ylabel('Firing Rate r(t)')
208     ax.legend()
209     ax.grid(True)
210
211 # Final layout
212 plt.suptitle("Tuning Curves Over Time for    =    ", fontsize=16)
213 plt.tight_layout(rect=[0, 0, 1, 0.96])
214 plt.show()
215
216
217 # %% [markdown]
218 # ## Question 2
219
220 # %%
221 # --- Build W    matrix (random symmetric)
222 W2, _, _, _ = network_model(2, m, phi, alpha, alpha_prime)
223
224 # --- Eigendecomposition of W
225 eigvals_W2, eigvecs_W2 = np.linalg.eig(W2)
226
227 # --- Plot 5 sample eigenvectors
228 plt.figure(figsize=(12, 6))
229 for i in range(5):
230     plt.plot(np.real(eigvecs_W2[:, i]), label=f"Eigenvector {i+1}"
        )
231 plt.title("Selected Eigenvectors of W    (Random Symmetric)")
232 plt.xlabel("Neuron index")
233 plt.ylabel("Component")

```

```

234 plt.legend()
235 plt.grid(True)
236 plt.tight_layout()
237 plt.show()
238
239
240 # %% [markdown]
241 # ## Question 3
242
243 # %%
244 # --- Get eigenvalues
245 eigvals_2 = np.linalg.eigvals(W2)
246 eigvals_3 = np.linalg.eigvals(W3)
247
248 # --- Plot sorted real parts of eigenvalues
249 plt.figure(figsize=(8, 5))
250 plt.plot(np.sort(np.real(eigvals_2)), label="W Spectrum")
251 plt.plot(np.sort(np.real(eigvals_3)), label="W Spectrum")
252 plt.xlabel("Index")
253 plt.ylabel("Eigenvalue (Real Part)")
254 plt.title("Spectral Comparison of W and W ")
255 plt.legend()
256 plt.grid(True)
257 plt.tight_layout()
258 plt.show()
259
260
261 # %%
262 # --- Eigendecomposition
263 eigvals_W2, eigvecs_W2 = np.linalg.eig(W2)
264 eigvals_W3, eigvecs_W3 = np.linalg.eig(W3)
265
266 # --- Sort by descending real eigenvalue (optional: for consistent
    comparison)
267 sorted_indices_W2 = np.argsort(-np.real(eigvals_W2))
268 sorted_indices_W3 = np.argsort(-np.real(eigvals_W3))
269
270 # --- Plot
271 fig, axs = plt.subplots(2, 5, figsize=(15, 6), sharex=True, sharey
    =True)
272

```

```

273 for i in range(5):
274     axs[0, i].plot(np.real(eigvecs_W2[:, sorted_indices_W2[i]]))
275     axs[0, i].set_title(f" W      Eigenvector {i+1}")
276     axs[1, i].plot(np.real(eigvecs_W3[:, sorted_indices_W3[i]]),
277                     color='green')
278     axs[1, i].set_title(f" W      Eigenvector {i+1}")
279     axs[0, i].set_ylabel("Component")
280     axs[1, i].set_ylabel("Component")
281
282 for ax in axs[1]:
283     ax.set_xlabel("Neuron index")
284
285 fig.suptitle("Top 5 Eigenvectors of W      (Random) vs W      (Ring)",
286             fontsize=16)
287 plt.tight_layout(rect=[0, 0, 1, 0.94])
288 plt.show()
289
290 # %% [markdown]
291 # ## Question 4
292
293 # %%
294 # --- Run everything for Model 1 and Model 2 ---
295 time_axis = np.linspace(0, duration, num_time_steps)
296 h_theta = h_theta_vector(phi, theta)
297
298 # --- Run everything for Models 1 to 4 ---
299 time_axis = np.linspace(0, duration, num_time_steps)
300 h_theta = h_theta_vector(phi, theta)
301
302 # Model 1
303 W1, B1, C1, _ = network_model(1, m, phi, alpha, alpha_prime)
304 r1 = simulate_model(W1, B1, C1, h_theta, tau, dt, num_time_steps)
305 error1 = compute_decoding_error(r1, C1, sigma, phi, theta,
306                                num_trials=500)
307
308 # Model 2
309 W2, B2, C2, _ = network_model(2, m, phi, alpha, alpha_prime)
310 r2 = simulate_model(W2, B2, C2, h_theta, tau, dt, num_time_steps)
311 error2 = compute_decoding_error(r2, C2, sigma, phi, theta,
312                                num_trials=500)

```

```

310
311 # Model 3
312 W3, B3, C3, _ = network_model(3, m, phi, alpha, alpha_prime)
313 r3 = simulate_model(W3, B3, C3, h_theta, tau, dt, num_time_steps)
314 error3 = compute_decoding_error(r3, C3, sigma, phi, theta,
    num_trials=500)
315
316 # Model 4
317 W4, B4, C4, _ = network_model(4, m, phi, alpha, alpha_prime)
318 r4 = simulate_model(W4, B4, C4, h_theta, tau, dt, num_time_steps)
319 error4 = compute_decoding_error(r4, C4, sigma, phi, theta,
    num_trials=500)
320
321 # --- Plot results ---
322 plt.figure(figsize=(10, 6))
323 plt.plot(time_axis, error1, label='Model 1: No Recurrence')
324 plt.plot(time_axis, error2, label='Model 2: Random Symmetric')
325 plt.plot(time_axis, error3, label='Model 3: Symmetric Ring')
326 plt.plot(time_axis, error4, label='Model 4: Balanced Ring')
327 plt.axhline(np.pi / 2, color='gray', linestyle=':', label='Chance
    level (  $\pi/2$  )')
328 plt.xlabel("Time (s)")
329 plt.ylabel("Average Decoding Error (rad)")
330 plt.title("Decoding Error Over Time (500 Trials)")
331 plt.legend()
332 plt.grid(True)
333 plt.tight_layout()
334 plt.show()
335
336
337
338 # %% [markdown]
339 # ## Question 5
340
341 # %%
342 # --- Question 5: Compare All Models + Model 4 with  $\alpha_{\text{prime}} = 5$  ---
343
344 # New alpha_prime value
345 alpha_prime_q5 = 5
346
347 # Rebuild Model 4 with  $\alpha_{\text{prime}} = 5$ 

```



```

348 W4_strong, B4_strong, C4_strong, _ = network_model(4, m, phi,
    alpha, alpha_prime_q5)
349 r4_strong = simulate_model(W4_strong, B4_strong, C4_strong,
    h_theta, tau, dt, num_time_steps)
350 error4_strong = compute_decoding_error(r4_strong, C4_strong, sigma
    , phi, theta, num_trials=500)
351
352 # --- Plot decoding error for all models including Model 4 (
    = 5) ---
353 plt.figure(figsize=(10, 6))
354 plt.plot(time_axis, error1, label='Model 1: No Recurrence')
355 plt.plot(time_axis, error2, label='Model 2: Random Symmetric')
356 plt.plot(time_axis, error3, label='Model 3: Symmetric Ring')
357 plt.plot(time_axis, error4, label='Model 4: Balanced Ring (
    = 0.9)')
358 plt.plot(time_axis, error4_strong, label='Model 4: Balanced Ring (
    = 5)', linestyle='--', linewidth=2)
359 plt.axhline(np.pi / 2, color='gray', linestyle=':', label='Chance
    level ( /2)')
360 plt.xlabel("Time (s)")
361 plt.ylabel("Average Decoding Error (rad)")
362 plt.title("Decoding Error Over Time (500 Trials,
    = )")
363 plt.legend()
364 plt.grid(True)
365 plt.tight_layout()
366 plt.show()
367
368
369 # %% [markdown]
370 # ## Question 6
371
372 # %%
373 # --- Question 6: Models 2 and 3 with
    = 5 ---
374 alpha_strong = 5 # New
375
376 # --- Model 2 with
    = 5 ---
377 W2_strong, B2_strong, C2_strong, _ = network_model(2, m, phi,
    alpha_strong, alpha_prime)
378 r2_strong = simulate_model(W2_strong, B2_strong, C2_strong,
    h_theta, tau, dt, num_time_steps)

```

```

379 error2_strong = compute_decoding_error(r2_strong, C2_strong, sigma
    , phi, theta, num_trials=500)
380
381 # --- Model 3 with      = 5 ---
382 W3_strong, B3_strong, C3_strong, _ = network_model(3, m, phi,
    alpha_strong, alpha_prime)
383 r3_strong = simulate_model(W3_strong, B3_strong, C3_strong,
    h_theta, tau, dt, num_time_steps)
384 error3_strong = compute_decoding_error(r3_strong, C3_strong, sigma
    , phi, theta, num_trials=500)
385
386 # --- Plot: Models 2 and 3 with      = 0.9 and      = 5 ---
387 plt.figure(figsize=(10, 6))
388 plt.plot(time_axis, error2, label='Model 2:      = 0.9')
389 plt.plot(time_axis, error2_strong, label='Model 2:      = 5',
    linestyle='--')
390 plt.plot(time_axis, error3, label='Model 3:      = 0.9')
391 plt.plot(time_axis, error3_strong, label='Model 3:      = 5',
    linestyle='--')
392 plt.axhline(np.pi / 2, color='gray', linestyle=':', label='Chance
    level ( /2)')
393 plt.xlabel("Time (s)")
394 plt.ylabel("Average Decoding Error (rad)")
395 plt.title("Q6: Effect of Increasing      on Models 2 and 3")
396 plt.legend()
397 plt.grid(True)
398 plt.tight_layout()
399 plt.show()
400
401
402 # %%
403 # --- Decoding Error Over Time (      = 0.9 vs      = 5) ---
404 plt.figure(figsize=(10, 4))
405 plt.plot(time_axis, error2, label='Model 2:      = 0.9', color='
    orange')
406 plt.plot(time_axis, error2_strong, label='Model 2:      = 5', color=
    'orange', linestyle='--')
407 plt.plot(time_axis, error3, label='Model 3:      = 0.9', color='
    green')
408 plt.plot(time_axis, error3_strong, label='Model 3:      = 5', color=
    'green', linestyle='--')

```

```

409 plt.axhline(np.pi / 2, color='gray', linestyle=':', label='Chance
    level (  $\pi/2$  )')
410 plt.xlabel("Time (s)")
411 plt.ylabel("Average Decoding Error (rad)")
412 plt.title("Decoding Error Over Time for Models 2 & 3")
413 plt.legend()
414 plt.grid(True)
415 plt.tight_layout()
416 plt.show()
417
418 # --- L2 Norm of r(t) over time for both models ---
419 norm_r2_strong = np.linalg.norm(r2_strong, axis=0)
420 norm_r3_strong = np.linalg.norm(r3_strong, axis=0)
421
422 plt.figure(figsize=(8, 4))
423 plt.plot(time_axis, norm_r2_strong, label="Model 2 (  $\sigma = 5$  )",
    color='orange')
424 plt.plot(time_axis, norm_r3_strong, label="Model 3 (  $\sigma = 5$  )",
    color='green')
425 plt.xlabel("Time (s)")
426 plt.ylabel("L2 Norm of  $r(t)$ ")
427 plt.title("L2 Norm of Neural Activity Over Time (  $\sigma = 5$  )")
428 plt.legend()
429 plt.grid(True)
430 plt.tight_layout()
431 plt.show()
432
433 # --- Tuning curves (population response at t = 60 ms) ---
434 idx_60 = int(0.06 / dt)
435
436 plt.figure(figsize=(12, 4))
437 plt.subplot(1, 2, 1)
438 plt.plot(phi, r2_strong[:, idx_60], color='orange')
439 plt.title("Model 2 (  $\sigma = 5$  )  $r(t=60\text{ms})$ ")
440 plt.xlabel("Preferred Orientation ")
441 plt.ylabel("Firing Rate")
442
443 plt.subplot(1, 2, 2)
444 plt.plot(phi, r3_strong[:, idx_60], color='green')
445 plt.title("Model 3 (  $\sigma = 5$  )  $r(t=60\text{ms})$ ")
446 plt.xlabel("Preferred Orientation ")

```

```

447 plt.ylabel("Firing Rate")
448
449 plt.suptitle("Q6: Population Response at t = 60ms")
450 plt.tight_layout(rect=[0, 0, 1, 0.95])
451 plt.show()
452
453 # --- Eigenvalue spectra of W matrices ---
454 eigvals2 = np.linalg.eigvals(W2_strong)
455 eigvals3 = np.linalg.eigvals(W3_strong)
456
457 plt.figure(figsize=(10, 4))
458 plt.subplot(1, 2, 1)
459 plt.plot(np.real(eigvals2), np.imag(eigvals2), 'o', label='Model 2
    ', color='orange')
460 plt.axvline(x=1, color='gray', linestyle='--')
461 plt.title("Model 2: Spectrum of  $W$  (  $\sigma = 5$  )")
462 plt.xlabel("Re( )")
463 plt.ylabel("Im( )")
464 plt.grid(True)
465
466 plt.subplot(1, 2, 2)
467 plt.plot(np.real(eigvals3), np.imag(eigvals3), 'o', label='Model 3
    ', color='green')
468 plt.axvline(x=1, color='gray', linestyle='--')
469 plt.title("Model 3: Spectrum of  $W$  (  $\sigma = 5$  )")
470 plt.xlabel("Re( )")
471 plt.ylabel("Im( )")
472 plt.grid(True)
473
474 plt.suptitle("Q6: Eigenvalue Spectra of W Matrices")
475 plt.tight_layout(rect=[0, 0, 1, 0.95])
476 plt.show()
477
478
479 # %%
480 fig, axs = plt.subplots(2, 2, figsize=(12, 8))
481
482 # --- Top row: r(t=60ms) tuning curves ---
483 axs[0, 0].plot(phi, r2_strong[:, idx_60], color='orange')
484 axs[0, 0].set_title("Model 2 (  $\sigma = 5$  )  $r(t=60ms)$ ")
485 axs[0, 0].set_xlabel("Preferred Orientation ")

```

```

486 axs[0, 0].set_ylabel("Firing Rate")
487
488 axs[0, 1].plot(phi, r3_strong[:, idx_60], color='green')
489 axs[0, 1].set_title("Model 3 (    = 5)      $r(t=60ms)$")
490 axs[0, 1].set_xlabel("Preferred Orientation    ")
491 axs[0, 1].set_ylabel("Firing Rate")
492
493 # --- Bottom row: Eigenvalue spectra ---
494 axs[1, 0].plot(np.real(eigvals2), np.imag(eigvals2), 'o', color='
    orange')
495 axs[1, 0].axvline(x=1, color='gray', linestyle='--')
496 axs[1, 0].set_title("Model 2: Spectrum of $W$ (    = 5)")
497 axs[1, 0].set_xlabel("Re(    )")
498 axs[1, 0].set_ylabel("Im(    )")
499 axs[1, 0].grid(True)
500
501 axs[1, 1].plot(np.real(eigvals3), np.imag(eigvals3), 'o', color='
    green')
502 axs[1, 1].axvline(x=1, color='gray', linestyle='--')
503 axs[1, 1].set_title("Model 3: Spectrum of $W$ (    = 5)")
504 axs[1, 1].set_xlabel("Re(    )")
505 axs[1, 1].set_ylabel("Im(    )")
506 axs[1, 1].grid(True)
507
508 fig.suptitle("Q6: Population Responses & Spectra for    = 5",
    fontsize=16)
509 plt.tight_layout(rect=[0, 0, 1, 0.95])
510 plt.show()
511
512
513 # %% [markdown]
514 # ## Question 7
515
516 # %%
517 # --- Model 5 Simulation ---
518 W5, B5, C5, _ = network_model(5, m, phi, alpha, alpha_prime)
519 r5 = simulate_model(W5, B5, C5, h_theta, tau, dt, num_time_steps)
520 error5 = compute_decoding_error(r5, C5, sigma, phi, theta,
    num_trials=500)
521
522 # --- Plot Model 4 vs Model 5 ---

```

```

523 plt.figure(figsize=(10, 6))
524 plt.plot(time_axis, error4, label='Model 4: Default B (      =
      0.9)')
525 plt.plot(time_axis, error5, label='Model 5: Balanced Input B',
      linestyle='--')
526 plt.axhline(np.pi / 2, color='gray', linestyle=':', label='Chance
      level ( /2)')
527 plt.xlabel("Time (s)")
528 plt.ylabel("Average Decoding Error (rad)")
529 plt.title("Q7: Improving Model 4 with Balanced Input")
530 plt.legend()
531 plt.grid(True)
532 plt.tight_layout()
533 plt.show()
534
535
536 # %% [markdown]
537 # ## Question 8
538
539 # %%
540 # --- Setup ---
541 alpha_primes = [0.5, 1, 2, 3, 5, 10, 25, 50]
542 decoding_errors = []
543 spectral_radaii = []
544 norms_at_t60 = []
545 variances_at_t60 = []
546 heatmaps_at_t60 = []
547
548 # --- Simulation parameters ---
549 h_theta = h_theta_vector(phi, theta)
550
551 for alpha_prime in alpha_primes:
552     W, B, C, _ = network_model(4, m, phi, alpha=0.9, alpha_prime=
        alpha_prime)
553     r = simulate_model(W, B, C, h_theta, tau, dt, num_time_steps)
554
555     # Decode error
556     err = compute_decoding_error(r, C, sigma, phi, theta,
        num_trials=100)
557     decoding_errors.append(err)
558

```

```

559     # Spectral radius
560     eigvals = np.linalg.eigvals(W)
561     spectral_radII.append(np.max(np.abs(eigvals)))
562
563     # Final timestep stats
564     final_r = r[:m, -1] # only excitatory population
565     norms_at_t60.append(np.linalg.norm(final_r))
566     variances_at_t60.append(np.var(final_r))
567     heatmaps_at_t60.append(final_r)
568
569
570 # %%
571 # 1. Decoding error over time
572 plt.figure(figsize=(10, 6))
573 for i, a in enumerate(alpha_primes):
574     plt.plot(time_axis, decoding_errors[i], label=f"      = {a}")
575 plt.axhline(np.pi/2, linestyle=':', color='gray', label='Chance (
    /2)')
576 plt.xlabel("Time (s)")
577 plt.ylabel("Decoding Error (rad)")
578 plt.title("Decoding Error vs Time for Increasing      (Model 4)")
579 plt.legend()
580 plt.grid(True)
581 plt.tight_layout()
582 plt.show()
583
584 # 2. Spectral radius
585 plt.figure(figsize=(6, 4))
586 plt.plot(alpha_primes, spectral_radII, marker='o')
587 plt.xlabel("      ")
588 plt.ylabel("Spectral Radius")
589 plt.title("Spectral Radius of W vs      ")
590 plt.grid(True)
591 plt.tight_layout()
592 plt.show()
593
594 # 3. Firing norm at t = 60ms
595 plt.figure(figsize=(6, 4))
596 plt.plot(alpha_primes, norms_at_t60, marker='o')
597 plt.xlabel("      ")
598 plt.ylabel("r (t=60ms) ")

```

```

599 plt.title("Firing Magnitude at Final Time")
600 plt.grid(True)
601 plt.tight_layout()
602 plt.show()
603
604 # 4. Variance of firing rates at t = 60ms
605 plt.figure(figsize=(6, 4))
606 plt.plot(alpha_primes, variances_at_t60, marker='o')
607 plt.xlabel(" ")
608 plt.ylabel("Var[r(t=60ms)]")
609 plt.title("Firing Rate Variance at Final Time")
610 plt.grid(True)
611 plt.tight_layout()
612 plt.show()
613
614 # 5. Heatmaps of firing activity at t = 60ms (with axis labels and
    colorbar)
615 fig, axs = plt.subplots(2, 3, figsize=(15, 6), sharey=True)
616 axs = axs.flatten()
617
618 # Use a common vmin/vmax for consistent color scaling
619 vmin = min([np.min(h) for h in heatmaps_at_t60])
620 vmax = max([np.max(h) for h in heatmaps_at_t60])
621
622 for i, a in enumerate(alpha_primes):
623     im = axs[i].imshow(heatmaps_at_t60[i][np.newaxis, :], aspect='
        auto',
624                        cmap='viridis', vmin=vmin, vmax=vmax)
625     axs[i].set_title(f" = {a}")
626     axs[i].set_yticks([0])
627     axs[i].set_yticklabels(["Firing"])
628     axs[i].set_xlabel("Neuron index")
629     axs[i].set_xlim(0, m)
630
631 # Add a shared colorbar
632 cbar_ax = fig.add_axes([0.92, 0.15, 0.015, 0.7])
633 fig.colorbar(im, cax=cbar_ax, label="Firing Rate")
634
635 fig.suptitle("Firing Rate Patterns at t = 60ms for Varying ",
    fontsize=16)
636 plt.tight_layout(rect=[0, 0, 0.9, 0.95])

```



```

637 plt.show()
638
639
640
641 # %%
642 # --- Extended Time Parameters ---
643 duration_long = 1.0 # 1000 ms
644 num_time_steps_long = int(duration_long / dt) + 1
645 time_axis_long = np.linspace(0, duration_long, num_time_steps_long
    )
646
647 # --- New simulate_model for long duration (non-destructive) ---
648 def simulate_model_long(W, B, C, h_theta, tau, dt, num_time_steps=
    num_time_steps_long):
649     n_model = W.shape[0]
650     r = np.zeros((n_model, num_time_steps))
651     r[:, 0] = B @ h_theta
652     for i in range(1, num_time_steps):
653         dr = (-r[:, i-1] + W @ r[:, i-1]) * (dt / tau)
654         r[:, i] = r[:, i-1] + dr
655     return r
656
657 # --- New compute_decoding_error for long duration (non-
    destructive) ---
658 def compute_decoding_error_long(r, C, sigma, phi, theta,
    num_trials=100):
659     m, T = C.shape[0], r.shape[1]
660     all_errors = []
661     for _ in range(num_trials):
662         trial_errors = []
663         for t in range(T):
664             noise = sigma * np.random.randn(m)
665             o_tilde = C @ r[:, t] + noise
666             x = np.sum(o_tilde * np.cos(phi))
667             y = np.sum(o_tilde * np.sin(phi))
668             theta_hat = np.arctan2(y, x) % (2 * np.pi)
669             error = np.abs(np.angle(np.exp(1j * (theta_hat - theta
                ))))
670             trial_errors.append(error)
671         all_errors.append(trial_errors)
672     return np.mean(all_errors, axis=0)

```

```

673
674
675 # %%
676 # --- Q8 Plot 4: Response to Weak/Noisy Stimulus ---
677
678 alpha_prime_low = 1
679 alpha_prime_high = 10
680 noise_strength = 1
681 num_trials = 100
682
683 # Use a noisy version of the stimulus
684 def noisy_h_theta(phi, theta, noise_scale=noise_strength):
685     return h_theta_vector(phi, theta) + noise_scale * np.random.
        randn(len(phi))
686
687 # Run Model 4 with noisy input for both values
688 errors_low = []
689 errors_high = []
690
691 for _ in range(num_trials):
692     h_noisy = noisy_h_theta(phi, theta)
693
694     # = 1
695     W_low, B_low, C_low, _ = network_model(4, m, phi, alpha=0.9,
        alpha_prime=alpha_prime_low)
696     r_low = simulate_model(W_low, B_low, C_low, h_noisy, tau, dt,
        num_time_steps_long)
697     err_low = compute_decoding_error(r_low, C_low, sigma, phi,
        theta, num_trials=1)
698     errors_low.append(err_low)
699
700     # = 10
701     W_high, B_high, C_high, _ = network_model(4, m, phi, alpha
        =0.9, alpha_prime=alpha_prime_high)
702     r_high = simulate_model(W_high, B_high, C_high, h_noisy, tau,
        dt, num_time_steps_long)
703     err_high = compute_decoding_error(r_high, C_high, sigma, phi,
        theta, num_trials=1)
704     errors_high.append(err_high)
705
706 # Average decoding errors across trials

```

```

707 avg_error_low = np.mean(errors_low, axis=0)
708 avg_error_high = np.mean(errors_high, axis=0)
709
710 # --- Plot ---
711 plt.figure(figsize=(10, 5))
712 plt.plot(time_axis, avg_error_low, label="          = 1")
713 plt.plot(time_axis, avg_error_high, label="          = 10", linestyle=
       "--")
714 plt.axhline(np.pi / 2, color="gray", linestyle=":", label="Chance
       level ( /2)")
715 plt.xlabel("Time (s)")
716 plt.ylabel("Average Decoding Error (rad)")
717 plt.title("Effect of          on Decoding Under Noisy Input (Model 4)
       ")
718 plt.legend()
719 plt.grid(True)
720 plt.tight_layout()
721 plt.show()
722
723
724 # %%
725 fig, axs = plt.subplots(2, 4, figsize=(15, 6), sharey=True)
726 axs = axs.flatten()
727
728 for i, a in enumerate(alpha_primes):
729     r_final = heatmaps_at_t60[i]
730     axs[i].plot(phi, r_final)
731     axs[i].set_title(f"          = {a}")
732     axs[i].set_xlabel("Preferred Orientation ")
733     if i % 4 == 0:
734         axs[i].set_ylabel("Firing Rate")
735
736 fig.suptitle("Model 4: Final Tuning Curves at t = 60ms for
       Different          ", fontsize=16)
737 plt.tight_layout(rect=[0, 0, 1, 0.95])
738 plt.show()

```