Mathematics for neuroscience - An overview

Lecture by Áine Byrne

October 25, 2021

***Computational Neuroscience, Neurotechnology and Neuro-inspired Artificial Intelligence* Autumn School**

Introduction

Mathematics has been of crucial importance in solving many historical challenges, particularly in the fields of physics and engineering, where mathematical concepts are regularly employed to address problems far beyond the context in which they were originally developed. More recently, mathematical models have been employed to solve problems in the realm of neuroscience, and biology more generally. Mathematical models of the brain make it possible to gain a deep and long-lasting insight into how the brain works.

In this lecture, we will review the key mathematical tools needed to develop such models of the brain. These tools include differential equations, linear algebra, and numerical analysis.

Hodgkin and Huxley

In 1963, Alan Hodgkin and Andrew Huxley were awarded the Nobel Prize in Physiology or Medicine for their discoveries concerning the ionic mechanisms involved in excitation and inhibition in the peripheral and central portions of the nerve cell membrane". Using a combination of electrophysiological recordings and mathematical intuition, they developed a mathematical description of how action potentials are initiate and propagate along a squid giant axon. Their work revolutionised neuroscience research and initiated a new field: mathematical neuroscience.

Hodgkin and Huxley's mathematical description consisted of four ordinary differential equations, prescribing the rate of change of the membrane potential (V) and the 3 additional quantities related to the potassium channel activation (n), sodium channel activation(m), and sodium channel inactivation (h).

$$C_m \frac{\mathrm{d}V}{\mathrm{d}t} = I - g_{Na} m^3 h (V - V_{Na}) - g_K n^4 (V - V_K) - g_l (V - V_l) \tag{1}$$

$$\tau_n(V) \frac{\mathrm{d}n}{\mathrm{d}t} = n_\infty(V) - n \tag{2}$$

$$\tau_m(V) \frac{\mathrm{d}m}{\mathrm{d}t} = m_\infty(V) - m \tag{3}$$

$$\tau_h(V) \frac{\mathrm{d}h}{\mathrm{d}t} = h_\infty(V) - h \tag{4}$$

The Wikipedia article on the Hodgkin and Huxley model provides a good overview of the model and its development.

Before we can study the equations of Hodgkin and Huxley, we must first learn about differential equations.

Differential equations

A differential equation prescribes the rate of change of a particular quantity. Consider the expression for unrestricted population growth

$$\frac{\mathrm{d}N}{\mathrm{d}t} = aN,$$

where N is the size of a given population, t is time and a is a parameter describing the growth rate. The derivative $\frac{\mathrm{d}N}{\mathrm{d}t}$ refers to the rate of change of N as t is varied, i.e. how is the population size going to change over time. On the right-hand side of the equation, we have an expression that prescribes that change. If a is a positive, our rate of change $\frac{\mathrm{d}N}{\mathrm{d}t}$ will be positive, i.e. the population size is going to increase. Whereas if a is a negative, the rate of change $\frac{\mathrm{d}N}{\mathrm{d}t}$ will be negative and the population size will decrease. Notice that there is also a N on the right-hand side of the equation. So, if N is small, the amount the population increases/decreases by is also going to be small, but the larger N gets the larger increase/decrease will become.

It may help to think about this in a *discretised* manner, e.g. how much do we expect the population to increase each year. Imagine the population of a particular town is 10,000 and it increases by $0.1 \times N$ each year. The change in population size this year will be $0.1 \times 10000 = 1000$, so next year the population size will be 11,000. Now applying the same logic, the population in two years time will be $11000 + 0.1 \times 11000 = 12100$. Below is a piece of code to apply this same logic to compute the population size for the next 25 years.

[ ]:
```
N = 10000
a = 0.1

print('Year   ','dN/dt    ','N')
for i in range(1,26):
    print(i, '     ',round(a*N), '    ', round(N+a*N))
    N = N + a*N
```

Exercise 1: What happens if we change a to be negative?

[ ]:
```

```

To study differential equations numerically, we need to manipulate and store arrays of numbers, which brings us to *linear algebra*.

Linear algebra

Linear algebra allows us to perform mathematical operations on arrays of numbers. Computational neuroscience, and computation more generally, relies heavily on linear algebra. The basic building block of linear algebra are vectors and matrices.

In this lecture will only cover the basics of linear algebra, if you would like to learn more, I recommend the Khan Academy course on linear algebra

Vectors

Vectors are essentially lists of numbers. Mathematically speaking, an n-dimensional vector (n numbers in the list) refers to a coordinate in n-dimensional space. For example, if we define a vector

$$v = \begin{bmatrix} x \\ y \end{bmatrix},$$

x is the amount we move in one direction and y is the amount we move in a perpendicular direction.

Python relies on a package called NumPy for linear algebra. Below is code for importing the NumPy package and creating a simple vector.

[ ]:
```
import numpy as np
```

[ ]:
```
v = np.array([1,2,3,4,5])
print(v)
```

In Python indexing starts a zero, so the first number in our vector is entry 0, the second is entry 1 and so on. To access specific entries, we use square brackets

[ ]:
```
v[2]
```

The colon operator can be used to access multiple entries. This is known as *slicing*. Run the code below to see how it works:

[ ]:
```
v[1:3]
```

[ ]:
```
v[2:]
```

[ ]:
```
v[:4]
```

Basic operations

Scalar multiplication A scalar is a single number, and scalar multiplication refers to multiplying a vector by a single number. With scalar multiplication, every entry is multiplied by this number.

[ ]:
```
v*2
```

Addition To add two vectors they must be the same length. The addition is performed element-by-element, i.e. the first element of vector one is added to the first element of vector two, the second element of vector one is added to the second element of vector two, and so on.

```
[ ]: u = np.array([3,7,1,6,4])
     v+u
```

Exercise 2: Add the vectors

$$a = \begin{bmatrix} 5 \\ 1 \\ 9 \\ 3 \\ 7 \end{bmatrix}, b = \begin{bmatrix} 7 \\ 3 \\ 1 \\ 4 \\ 6 \end{bmatrix}$$

by hand and then use Python to check your answer.

```
[ ]:
```

Dot product The dot product of two vectors is computed by performing element-by element multiplication and adding up all of the products.

$$u \cdot v = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n.$$

As with adding two vectors, the two vectors must be the same length. To compute the dot product in Python we use the `dot()` function/method.

```
[ ]: v.dot(u)
```

Note: Using `v*u` will perform element-by-element multiplication, but not sum up the products.

Exercise 3: Compute the dot product of the vectors a and b (given in Exercise 2) by hand. Then compute the dot product in Python. Do the two answers match?

```
[ ]:
```

Matrices

A matrix can be thought of as a collection of vectors of the same length. An $n \times m$ matrix is a rectangular array of numbers with n rows and m columns. For example,

$$A = \begin{bmatrix} 2 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 1 & 6 \\ 8 & 3 & 5 \end{bmatrix} \text{ is a } 4 \times 3 \text{ matrix, while } B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \text{ is a } 3 \times 2 \text{ matrix.}$$

In Python, matrices are create in a similar manner to vectors. We simply give the `array` function a list of lists

[ ]:
```
A = np.array([[2,8,4],[1,0,3],[5,1,6],[8,3,5]])
print(A)
```

[ ]:
```
B = np.array([[1,2],[3,4],[5,6]])
print(B)
```

[ ]:
```
A[0,1]
```

[ ]:
```
B[2,1]
```

A common use of matrices is to digitally encode a picture. Imagine a 100×100 pixel black and white image. Each pixel encodes the level of brightness at the point, which is just a single number. Writing down the brightness level at each pixel in a 100×100 grid gives us a 100×100 matrix.

[ ]:
```
pixel_matrix = np.load('pixel_matrix.npy')
print(pixel_matrix)
print(pixel_matrix.shape)
```

Let's try plotting the matrix to see what it represents. We will first need to load in Python's plotting library matplotlib:

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

[ ]:
```
plt.matshow(pixel_matrix,cmap='gray')
```

Basic operations

Scalar multiplication As with vectors, if we multiply a matrix by a scalar, we simply multiple every entry in the matrix by that number.

[ ]:
```
A*3
```

Matrix-vector multiplication We can multiple a $n \times m$ matrix (A) by a m-dimensional vector (x) and the result will be a n-dimensional vector. To perform this multiplication we compute the dot product of each of the rows of A with x. The result is a vector with m entries, where the first entry is dot product of the first row of A with x, the second entry is dot product of the second row of A with x, and so on.

[ ]:
```
x = np.array([[2],[4],[1]])
A.dot(x)
```

Note: The order of multiplication matters! We cannot multiple a m-dimensional vector by a $n \times m$ matrix. The number of columns in the first matrix/vector must be the same as the number of rows in the second matrix/vector.

[ ]:
```
x.dot(A)
```

Exercise 4: Compute the product Mu, where

$$M = \begin{bmatrix} 1 & 7 & 3 \\ 9 & 6 & 7 \end{bmatrix} \text{ and } U = \begin{bmatrix} 2 \\ 9 \\ 6 \end{bmatrix}$$

by hand and then use Python to check your answer.

[ ]:

Matrix multiplication Matrix multiplication is simply an extension of matrix-vector multiplication. When multiplying two matrices we compute the dot product of each row of matrix 1 with each column of matrix 2, and the resulting matrix contains all of these dot products.

Lets take our matrices A and B from above and compute their product:

[ ]:
```
A.dot(B)
```

Multiplying a $n \times m$ matrix by a $m \times p$ matrix results in a $n \times p$ matrix (A).

Note: The number of columns in the first matrix must be the same as the number of rows in the second matrix. Hence, the product BA is not defined, as B has 2 columns and A has 4 rows.

Exercise 5: Compute the product XY, where

$$X = \begin{bmatrix} 5 & 1 \\ 6 & 3 \end{bmatrix} \text{ and } Y = \begin{bmatrix} 3 & 7 & 2 \\ 1 & 4 & 6 \end{bmatrix}$$

by hand and then use Python to check your answer.

[ ]:

Solving a linear system Suppose we are given the system of equations

$$2x + y - 7z = 16x - 2y - z = 85x - 3y + 4z = 7$$

and asked to solve them for x, y and z.

This system of equations can be written in matrix form as

$$\begin{bmatrix} 2 & 1 & 7 \\ 6 & -2 & -1 \\ 5 & -3 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 8 \\ 7 \end{bmatrix}$$

Then we can use the `solve` function from linear algebra module (`linalg`) of NumPy to solve this matrix equation.

[ ]:
```
M_eq =  np.array([[2, 1, -7], [6, -2, -1], [5, -3, 4]])
v_eq = np.array([[1], [8], [7]])
solution = np.linalg.solve(M_eq,v_eq)
```

```
print(solution)
```

You can manually check the result by plugging in $x = 4$, $y = 7$ and $z = 2$ into the system of equations, or by multiplying the matrix by the solution and verifying that it matches the vector on the right-hand side.

[ ]:
```
M_eq.dot(solution)
```

Solving differential equations

Now that we know how to store numbers in arrays and perform basic manipulations on these arrays, we can develop tools for studying differential equations.

Recall our differential equation for unrestricted population growth

$$\frac{\mathrm{d}N}{\mathrm{d}t} = aN.$$

This type of equation is called an ordinary differential equation, or ODE for short.

This equation can be solved analytically by separating the derivative $\frac{\mathrm{d}N}{\mathrm{d}t}$ and integrating both sides:

$$\int_{N_0}^{N} \frac{1}{N'} \mathrm{d}'N = \int_0^t a \mathrm{d}t' \tag{5}$$

$$\log(N) - \log(N_0) = at \tag{6}$$

$$N(t) = N_0 * \mathrm{e}^{at} \tag{7}$$

See Ordinary differential equation examples on Maths Insight for details on how to solce ODEs analytically. Maths is Fun also have a nice tutorial on First Order Linear Differential Equations.

Setting our initial population size N_0 and growth rate a we can compute the population size at all points in the future:

[ ]:
```
N0 = 10000
a = 0.1
t = np.linspace(0,25,101)
N = N0*np.exp(a*t)
```

Let's plot the solution to see how the population size evolves with time

[ ]:
```
plt.figure()
plt.plot(t,N)
plt.xlabel('Time (years)')
plt.ylabel('Population size')
plt.axis([0,25,10000,120000])
```

Unfortunately, most ODEs do not have nice analytic solutions and we are forced to rely on numerical estimations. See the Wikipedia page Numerical_methods_for_ordinary_differential_equation for an overview of the different numerical methods for solving ODEs.

Euler's method

Euler's method is the simplest numerical method for estimating the solution of a differential equation. Similar to when we *discretised* the equation for unrestricted population growth and asked what the population increase was each year, Euler's method takes small time steps and computes the increase at each time step. The smaller the time step is, the more accurate the solution.

$$x_{n+1} = x_n + hf(x_n, y_n)$$

For the model of unrestricted population growth, the Euler method is defined as

$$N_{n+1} = N_n + \Delta t \times aN_n,$$

where Δt is the length of the time step we take.

The first thing we should do is define a function to represent the right-hand side of our equation:

[ ]:
```
def dNdt(a,x):
    return a*x
```

Next, we set up the Euler's method and cycle through our time points, estimating the solution at each timepoint.

[ ]:
```
dt = 1
t_est = np.arange(0,25+dt,dt)
N_est = np.zeros(t_est.size)
N_est[0] = N0
for j in range(len(t_est)-1):
    N_est[j+1] = N_est[j] + dt*dNdt(a,N_est[j])
```

Now we plot the estimated solution and the exact solution on the same graph.

[ ]:
```
plt.figure()
plt.plot(t_est,N_est,label='Estimate')
plt.plot(t,N,label='Exact')
plt.xlabel('Time (years)')
plt.ylabel('Population size')
plt.axis([0,25,N0,121000])
plt.legend()
```

Examining the graph, we see that the estimated solution underestimates the true solution and that it becomes more and more inaccurate as time increases.

Exercise 6: Change the step size Δt to 0.1 and run the simulation again. What do you observe?

[ ]:
```

```

Runge-Kutta method

We can improve the accuracy of our numerical solver by including higher order term. However, higher order methods require more calculations and function evaluations, and as such, will take longer to run. In practice, a good balance is achieved by the fourth order Runge–Kutta method. Instead of simply using our current population size to estimate, say next years population, we use estimates throughout the time interval (the year) to determine our estimate. The solution at each time point is computed as follows:

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where

$$k_1 = \Delta t f(x_n) \tag{8}$$
$$k_2 = \Delta t f(x_n + \tfrac{1}{2}k_1) \tag{9}$$
$$k_3 = \Delta t f(x_n + \tfrac{1}{2}k_2) \tag{10}$$
$$k_4 = \Delta t f(x_n + k_3) \tag{11}$$

See Harold Serrano's blog post for a nice visualisation of the Runge-Kutta method.

As we did for Euler's method, we set up the Runge-Kutta method and iterate over time to compute the estimated solution.

[ ]:
```
dt = 1
N_RK = np.zeros(t_est.size)
N_RK[0] = N0
for n in range(len(t_est)-1):
    k1 = dt*dNdt(a,N_RK[n])
    k2 = dt*dNdt(a,N_RK[n]+0.5*k1)
    k3 = dt*dNdt(a,N_RK[n]+0.5*k2)
    k4 = dt*dNdt(a,N_RK[n]+k3)
    N_RK[n+1] = N_RK[n] + (k1 + 2*k2 + 2*k3 + k4)/6
```

Now plotting the exact solutions and the two estimated solutions.

[ ]:
```
plt.figure()
plt.plot(t,N,label='Exact')
plt.plot(t_est,N_est,label='Euler')
plt.plot(t_est,N_RK,label='Runge Kutta')

plt.xlabel('Time (years)')
plt.ylabel('Population size')
plt.legend()
```

Looking at the plot, we see that if we use the same time step for both the Euler method and the Runge-Kutta method, the Runge-Kutta method significantly outperforms the Euler method.

Built-in ODE solvers

In practice, we usually rely on built-in ODE solvers. They are tried and tested functions that will solve a system of ODEs to a high degree of accuracy. These functions will typically be more efficient that one you write yourself.

Scipy's integrate module contains an array of functions for numerical calculus (numerical differentiation, numerical integration, ODE solvers etc.). You can find a list of all of the ODE solvers included in the integrate module here

We will focus on the solve_ivp function as it allows us choose from a list of integration methods. The default is the 4th order Runge-Kutta method, with an adaptive step size (updates the step size throughout the simulation, balancing accuracy and efficiency).

[ ]:
```
from scipy.integrate import solve_ivp
```

To use any of the ODE solvers from Scipy's integrate module we must define a function where the first two input arguments are time t and the variable/list of variables x, in that order. The system parameter can then be defined as additional input arguments

[ ]:
```
def dNdt_ivp(t,x,a):
    return a*x
```

The syntax for solve_ivp is `(function name, [t start, t end], initial conditions, args)`. You can include additional arguments to specify the integration method, the tolerance, step size, etc. Read the function documentation for more details.

[ ]:
```
sol = solve_ivp(dNdt_ivp, [0, 25], [N0], args=(a,), dense_output=True)
```

We can evaluate the solution for an array of time points and plot the solution:

[ ]:
```
t_ivp = np.linspace(0, 25, 101)
N_ivp = sol.sol(t_ivp)

plt.figure(figsize=(12,6))
plt.subplot(1,2,1)
plt.plot(t,N,label='Exact')
plt.plot(t_est,N_est,label='Euler')
plt.plot(t_est,N_RK,label='Runge Kutta')
plt.plot(t_ivp,N_ivp[0],label='Built-in')

plt.xlabel('Time (years)')
plt.ylabel('Population size')
plt.legend()


plt.subplot(1,2,2)
plt.plot(t,N,label='Exact')
plt.plot(t_est,N_RK,label='Runge Kutta')
```

```
plt.plot(t_ivp,N_ivp[0],label='Built-in')

plt.xlabel('Time (years)')
plt.axis([24.8,25,119000,122000])
```

Note: we need to include `dense_output=True` as an additional argument to evaluate the solution on a dense mesh.

Exercise 7: Consider a system described by the following ODE

$$\dot{x} = x(1-x).$$

Define a function `dxdt` for the right-hand side of this expression, using the notation specied above (first two arguments must be t and x). Then use the `solve_ivp` function to estimate the solution on the interval $0 \leq t \leq 10$ and initial value $x(0) = 0.5$.

[ ]:

Simulating the Hodgkin Huxley model

Now that we understand what ODEs are and how we can simulate them computationally, we return to the model of Hodgkin and Huxley:

$$C_m \frac{\mathrm{d}V}{\mathrm{d}t} = I - g_{Na} m^3 h (V - V_{Na}) - g_K n^4 (V - V_K) - g_l (V - V_l) \tag{12}$$

$$\frac{\mathrm{d}n}{\mathrm{d}t} = \alpha_n(V)(1-n) - \beta_n(V) n \tag{13}$$

$$\frac{\mathrm{d}m}{\mathrm{d}t} = \alpha_m(V)(1-m) - \beta_m(V) m \tag{14}$$

$$\frac{\mathrm{d}h}{\mathrm{d}t} = \alpha_h(V)(1-h) - \beta_h(V) h \tag{15}$$

First, we set up a function that takes the variables (V, n, m, h) as a vector and then returns the right-hand side of each of the equations, again as a vector. We will vary the input current I. Hence, why it is included as an input argument.

[ ]:
```
def HH(t,x,I):

    V = x[0]
    n = x[1]
    m = x[2]
    h = x[3]

    alpha_n = 0.02*(V-25) / (1 - np.exp(-(V-25)/9) )
    alpha_m = 0.182*(V+35) / (1 - np.exp(-(V+35)/9))
    alpha_h = 0.25*np.exp(-(V+90)/12)

    beta_n = -0.002*(V-25)/(1-np.exp((V-25)/9))
```

```
    beta_m = -0.124*(V+35)/(1-np.exp((V+35)/9))
    beta_h = 0.25*np.exp((V+62)/6) / np.exp((V+90)/12)

    dVdt = 1/C*(I - gNa*m**3*h*(V-VNa) - gK*n**4*(V-VK) - gL*(V-VL))
    dndt = alpha_n*(1-n) - beta_n*n
    dmdt = alpha_m*(1-m) - beta_m*m
    dhdt = alpha_h*(1-h) - beta_h*h

    return [dVdt, dndt, dmdt, dhdt]
```

Now we define the parameters and simulate the model using the `solve_ivp` function

[ ]:
```
# Define parameters
C = 1
gNa = 40
gK = 35
gL = 0.3
VNa = 55
VK = -77
VL = -65

# Simulate model
HH_sol = solve_ivp(HH, [0,100], [-60,0,0,0.6], dense_output = True, args = (0.
↪2,))
t_HH = np.linspace(0, 100, 1000)
x_HH = HH_sol.sol(t_HH)
plt.plot(t_HH, x_HH[0])
plt.xlabel('Time')
plt.ylabel('Voltage')
```

Exercise 8: Vary the current I (the input argument) and describe how the dynamics of the voltage change as I is increased

[ ]:
```

```

Kuramoto model

Once we begin to look at large populations of neurons, simulating a detailed model, such as the Hodgkin-Huxley model, becomes computationally expensive. Instead, we opt for simple caricatures that capture the important elements of a neuron's dynamics without keeping track of every detail. The Kuramoto model is one such model. It is particularly useful if we wish to understand how neurons *synchronise* their firing times.

The Kuramoto model describes the evolution of a neuron's *phase*, how close it is to spiking. The

simplest form of the model is given as follows:

$$\frac{\mathrm{d}\theta_i}{\mathrm{d}t} = \omega_i + \frac{k}{N}\sum_{j=1}^{N}\sin(\theta_j - \theta_i),$$

where θ_i is the phase of the ith neuron, ω_i is the intrinsic natural frequency of the ith neuron, k is the coupling strength and N is the number of neurons in the population. The Wikipedia article on the Kuramoto model provides an nice overview of the model and some of its variations.

As before, we set up a function defining our system:

[ ]:
```
def kuramoto(t,x,k):

    phase_diff = np.subtract.outer(x, x)

    return omega + (k/N) * (np.sin(-phase_diff)).sum(axis=1)
```

We will need SciPy's statistics module to create our distribution of intrinsic frequencies. We choose the intrinsic frequencies from a Lorentzian with centre ω_0 and width at half maximum Δ

[ ]:
```
import scipy.stats as stats
```

[ ]:
```
# Define parameters
k = 1
N = 100
omega0 = 0.5
Delta = 0.01
omega = stats.cauchy.rvs(loc=omega0, scale=Delta, size=N)
theta0 = 2*np.pi*np.random.rand(N)

# Simulate model
kuramoto_sol = solve_ivp(kuramoto, [0,100], theta0, dense_output = True, args =␣
↪(k,))
t_K = np.linspace(0, 100, 1000)
x_K = kuramoto_sol.sol(t_K)%(2*np.pi)

# Plot
plt.figure()
plt.plot(t_K, x_K.T)
plt.xlabel('Time')
plt.ylabel('Phase')
```

A nice way to visualise the dynamics of Kuramoto oscillators, is to pose them on a circle and watch their phases evolve. To do this, we create an animated plot

[ ]:
```
# Import module for animations
import matplotlib.animation as ani
# Configure Jupyter for animated plots
%matplotlib notebook
```

I've copied the code for simulating the Kuramoto model down here so that we can easily re-run it for different coupling strengths and see how this affects the dynamics

[ ]:
```
# Simulate the Kuramoto model
k = 0
kuramoto_sol = solve_ivp(kuramoto, [0,100], theta0, dense_output = True, args =␣
↪(k,))
x_K = kuramoto_sol.sol(t_K)%(2*np.pi)
```

[ ]:
```
# Set up animation
theta=np.linspace(0,2*np.pi,1000)
fig, ax = plt.subplots(figsize=(6,6))

ax.set_xlim((-1.1, 1.1))
ax.set_ylim((-1.1, 1.1))
line, = ax.plot([], [], 'b.', lw=2, markersize=10)

def init():
    plt.plot(np.cos(theta),np.sin(theta),'k--',linewidth=0.8)
    line.set_data([], [])
    return (line,)

def dots(i):
    x = np.cos(x_K[:,i])
    y = np.sin(x_K[:,i])
    line.set_data(x, y)
    return (line,)

# Create animation
animator = ani.FuncAnimation(fig, dots, init_func=init, interval=100,␣
↪frames=range(len(t_K)), blit=True, repeat=False)
plt.show()
```

For weak coupling, the neurons continue to act independently. As we increase the coupling strength, some of the slower neurons will speed up and some of the faster neurons will slow down. This results in the neurons *synchronising* their activity. For very large values of k all the neurons will synchronise, evolving at the same rate and phase.

Kuramoto order parameter

Computing the *order parameter* allows us to quantify how synchronous or asynchronous the population of neurons is.

$$Z = Re^{i\Psi} = \frac{1}{N}\sum_{j=1}^{N} e^{i\theta_j}.$$

The order parameter is a complex number, whose magnitude R corresponds to the level of synchrony and phase Ψ is the average phase of the population of neurons.

[ ]:
```
# Simulate the Kuramoto model
k = 1
kuramoto_sol = solve_ivp(kuramoto, [0,100], theta0, dense_output = True, args =␣
 ↪(k,))
x_K = kuramoto_sol.sol(t_K)%(2*np.pi)

# Compute the Kuramoto order parameter
Z = 1/N * np.exp(complex(0, 1) * x_K).sum(axis=0)
```

Now we create an animation that shows the dynamics of the individual neurons as well as the Kuramoto order parameter.

[ ]:
```
# Set up animation
fig, ax = plt.subplots(figsize=(6,6))

ax.set_xlim((-1.1, 1.1))
ax.set_ylim((-1.1, 1.1))

line1, = ax.plot([], [], 'b.', lw=2, markersize=10)
line2, = ax.plot([], [], color=[0.7,0.7,0.7])
dot, = ax.plot([], [], '*', color=[0.7,0.7,0.7], lw=2, markersize=10)

def init():
    plt.plot(np.cos(theta),np.sin(theta),'k--',linewidth=0.8)
    line1.set_data([], [])
    line2.set_data([], [])
    dot.set_data([], [])
    return (line1,line2,dot,)

def dots(i):
    x =  np.cos(x_K[:,i])
    y =  np.sin(x_K[:,i])
    line1.set_data(x,y)
    line2.set_data(Z[:i].real, Z[:i].imag)
    dot.set_data(Z[i].real, Z[i].imag)
    return (line1,line2,dot,)

# Create animation
animator = ani.FuncAnimation(fig, dots, init_func=init, interval=100,␣
 ↪frames=range(len(t_K)), blit=True, repeat=False)
plt.show()
```

Ott-Antonsen reduction

The beauty of the Kuramoto model is that it is amenable to an exact mean field reduction. In the large N limit ($N \to \infty$) the population dynamics can be captured by a single equation that

describes the evolution of the order parameter:

$$\frac{dZ}{dt} = (-\Delta + i\omega_0)Z + \frac{1}{2}kZ(1 - |Z|^2),$$

where the intrinsic frequencies ω_o are chosen from a Lorentzian distribution with centre ω_0 and width at half maximum Δ.

This reduction was derived by Ott and Antonsen in 2008, and has since been shown to extend to many other (more complex) phase oscillator models.

Remembering that $Z = Re^{i\Psi}$, we arrive at equations for the synchrony R and average phase Ψ:

$$\frac{dR}{dt} = \left[-\Delta + \frac{1}{2}k(1 - R^2)\right] R \tag{16}$$

$$\frac{d\Psi}{dt} = \omega_0 \tag{17}$$

[ ]:
```
# Define function describing the Kuramoto order parameter in the mean field limit
def kuramoto_MF(t,x,k):

    R = x[0]
    Psi = x[1]

    dR = (-Delta + 1/2 * k * (1-R**2))*R
    dPsi = omega0

    return (dR, dPsi)
```

[ ]:
```
# Simulate the mean-field verion of the Kuramoto model
kuramoto_sol = solve_ivp(kuramoto_MF, [0,100], (abs(Z[0]),np.angle(Z[0])),␣
 ↪dense_output = True, args = (k,))
x_K = kuramoto_sol.sol(t_K)
R = x_K[0,:]
Psi = x_K[1,:]
```

Plotting the mean-field dynamics of the order parameter and the order parameter computed for the simulation of 100 neurons, we see that two are closely matched.

[ ]:
```
plt.figure(figsize=(5,5))
plt.plot(R*np.cos(Psi),R*np.sin(Psi),label='Mean-field')
plt.plot(Z.real,Z.imag,label='100 neurons')
plt.xlabel('Real(Z)')
plt.ylabel('Imag(Z)')
plt.legend(loc='upper left')
```

Synaptic plasticity

Action potentials are transmitted from one neuron (presynaptic) to another (postsynaptic) at a synapse. When the action potential reaches the synapse the presynaptic neuron releases neurotransmitter, which causes passages to open between the two neurons. These passages are known as ion channels and once open ions can flow in and out of the postsynaptic neuron. Ions have electrical charge, so ions flowing in and out of the neuron is going to lead to a change in voltage. Once the voltage of the postsynaptic neuron increases past a certain value it will release its own action potential and the cycle repeats.

Synaptic plasticity refers to the activity-dependent modification of the strength or efficacy of a synapses, i.e. how much of a change in voltage can one action potential create. Such changes in synaptic strength can modify the function of a neural circuit, enabling us to learn and creating memories. See Citri & Malenka (2007) for an overview of synaptic plasticity

We will focus on two forms of synaptic plasticity: Hebbian learning and spike-time dependent plasticity (STDP).

Hebbian learning

In the 1940s Donald Hebb postulated that *cells that fire together wire together*. The general idea of his theory was that if neuron A and neuron B are regularly activated at the same time, then the synaptic connection between them should gradually become more effective. While if neuron A and neuron B's firing pattern are uncorrelated, the synapse should become less effective.

Hebb's theory was later formalised mathematically and there now exist many formulations of Hebbian learning, see Gerstner & Kistler (2002) for a nice review.

We can create a simple caricature of Hebbian learning, by assigning different coupling strengths k_{ij} for each of the connections and defining a rule that update these coupling strengths based on the phase difference:

$$\frac{\mathrm{d}\theta_i}{\mathrm{d}t} = \omega_i + \frac{1}{N}\sum_{j=1}^{N} k_{ij}\sin(\theta_j - \theta_i), \tag{18}$$

$$\frac{\mathrm{d}k_{ij}}{\mathrm{d}t} = \epsilon(\alpha\cos(\theta_j - \theta_i) - k_{ij}). \tag{19}$$

When the phase difference is small, the coupling strength will increase, while if it is large the coupling strength will decrease.

[ ]:
```
def kuramoto_plasticity(t,x):

    theta = x[:N]
    k = x[N:].reshape(N,N)

    phase_diff = np.subtract.outer(theta, theta)

    dtheta = omega + (1/N) * (k*np.sin(-phase_diff)).sum(axis=1)
    dk = epsilon * ( alpha*np.cos(phase_diff) - k )
```

```
    return np.concatenate((dtheta, dk.reshape(N**2,)))
```

[ ]:
```
# Define parameters
epsilon = 0.5
alpha = 1
N = 50
omega = stats.cauchy.rvs(loc=0.5, scale=0.01, size=N)
theta0 = 2*np.pi*np.random.rand(N)
k0 = np.random.rand(N,N)
x0 = np.concatenate((theta0,k0.reshape(N**2,)))

# Simulate
kuramoto_sol = solve_ivp(kuramoto_plasticity, [0,100], x0, dense_output = True)
t_K = np.linspace(0, 100, 1000)
theta_K = kuramoto_sol.sol(t_K)[:N,:]%(2*np.pi)
k_ij = kuramoto_sol.sol(t_K)[N:,:]

# Compute the Kuramoto order parameter
Z = 1/N * np.exp(complex(0, 1) * theta_K).sum(axis=0)

# Plot
plt.figure()
plt.subplot(2,1,1)
plt.plot(t_K,theta_K.T)
plt.ylabel('Phase')

plt.subplot(2,1,2)
plt.plot(t_K,k_ij.T)
plt.xlabel('Time')
plt.ylabel('Coupling strength')
```

[ ]:
```
# Set up animation
fig, ax = plt.subplots(figsize=(6,6))

ax.set_xlim((-1.1, 1.1))
ax.set_ylim((-1.1, 1.1))

line1, = ax.plot([], [], 'b.', lw=2, markersize=10)
line2, = ax.plot([], [], color=[0.7,0.7,0.7])
dot, = ax.plot([], [], '*', color=[0.7,0.7,0.7], lw=2, markersize=10)

def init():
    plt.plot(np.cos(theta),np.sin(theta),'k--',linewidth=0.8)
    line1.set_data([], [])
    line2.set_data([], [])
    dot.set_data([], [])
```

```
    return (line1,line2,dot,)

def dots(i):
    x =  np.cos(theta_K[:N,i])
    y =  np.sin(theta_K[:N,i])
    line1.set_data(x,y)
    line2.set_data(Z[:i].real, Z[:i].imag)
    dot.set_data(Z[i].real, Z[i].imag)
    return (line1,line2,dot,)

# Create animation
animator = ani.FuncAnimation(fig, dots, init_func=init, interval=100,␣
 ↪frames=range(len(t_K)), blit=True, repeat=False)
plt.show()
```

Spike-time dependent plasticity (STDP)

Spike-time dependent plasticity is an asymmetric version of Hebbian learning. Instead of both synapses ($A \to B$ and $B \to A$) being strengthened when A and B fire together, one is strengthened and the other is weakened, depending on the order of the spikes. If A spikes and shortly afterward B spikes, the link from A to B is said to be causal and as such the $A \to B$ synapse is strengthen while the $B \to A$ synapse is weakened as A firing is not causally related to B firing.

The scholarpedia Spike-time dependent plasticity provides a nice summary of the different mathematical models of STDP.

Again, for a simple caricature of STDP we use the same formalism as above, but replace the cos in the learning rule with a sin:

$$\frac{\mathrm{d}\theta_i}{\mathrm{d}t} = \omega_i + \frac{1}{N}\sum_{j=1}^{N} k_{ij}\sin(\theta_j - \theta_i), \tag{20}$$

$$\frac{\mathrm{d}k_{ij}}{\mathrm{d}t} = \epsilon(\alpha\sin(\theta_j - \theta_i) - k_{ij}). \tag{21}$$

Note: This is a grossly over simplified version of STDP. In particular, with STDP the maximal update of weights should be when the spike-times are very similar. Using a sin function means that the maximal update when the phase difference is $\pm\pi/2$.

[ ]:
```
def kuramoto_STDP(t,x):

    theta = x[:N]
    k = x[N:].reshape(N,N)

    phase_diff = np.subtract.outer(theta, theta)

    dtheta = omega + (1/N) * (k*np.sin(-phase_diff)).sum(axis=1)
    dk = epsilon * ( alpha*np.sin(-phase_diff) - k )
```

```
    return np.concatenate((dtheta, dk.reshape(N**2,)))
```

[ ]:
```
# Define parameters
epsilon = 0.5
alpha = 1
N = 50
omega = stats.cauchy.rvs(loc=0.5, scale=0.1, size=N)
theta0 = 2*np.pi*np.random.rand(N)
k0 = np.random.rand(N,N)
x0 = np.concatenate((theta0,k0.reshape(N**2,)))

# Simulate
kuramoto_sol = solve_ivp(kuramoto_STDP, [0,100], x0, dense_output = True)
t_K = np.linspace(0, 100, 1000)
theta_K = kuramoto_sol.sol(t_K)[:N,:]%(2*np.pi)
k_ij = kuramoto_sol.sol(t_K)[N:,:]

# Compute the Kuramoto order parameter
Z = 1/N * np.exp(complex(0, 1) * theta_K).sum(axis=0)

# Plot
plt.figure()
plt.subplot(2,1,1)
plt.plot(t_K,theta_K.T)
plt.ylabel('Phase')

plt.subplot(2,1,2)
plt.plot(t_K,k_ij.T)
plt.xlabel('Time')
plt.ylabel('Coupling strength')
```

[ ]:
```
# Set up animation
fig, ax = plt.subplots(figsize=(6,6))

ax.set_xlim((-1.1, 1.1))
ax.set_ylim((-1.1, 1.1))

line1, = ax.plot([], [], 'b.', lw=2, markersize=10)
line2, = ax.plot([], [], color=[0.7,0.7,0.7])
dot, = ax.plot([], [], '*', color=[0.7,0.7,0.7], lw=2, markersize=10)

def init():
    plt.plot(np.cos(theta),np.sin(theta),'k--',linewidth=0.8)
    line1.set_data([], [])
    line2.set_data([], [])
    dot.set_data([], [])
```

```
    return (line1,line2,dot,)

def dots(i):
    x =  np.cos(theta_K[:N,i])
    y =  np.sin(theta_K[:N,i])
    line1.set_data(x,y)
    line2.set_data(Z[:i].real, Z[:i].imag)
    dot.set_data(Z[i].real, Z[i].imag)
    return (line1,line2,dot,)

# Create animation
animator = ani.FuncAnimation(fig, dots, init_func=init, interval=100,␣
 ↪frames=range(len(t_K)), blit=True, repeat=False)
plt.show()
```

[ ]:

[ ]:

[ ]: