Mathematics for Neuroscience - An introduction

1 Mathematics for Neuroscience - An Introduction

Lecture by Rachel Nicks

Notes by Áine Byrne and Rachel Nicks

October 25, 2023

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

2 Introduction

Despite the immense complexity of the brain, mathematical modelling has allowed for major advances to be made towards understanding behaviour, consciousness and disease. Mathematical models can be used to describe processes from the level of single cell voltage dynamics, through emergent behaviour of neural networks to activity patterns in tissue level models. Underlying nearly all of these models are differential equations describing how various quantities (e.g. voltage, firing rate) change in time and space. This lecture introduces some of the mathematical tools needed to understand and analyse solutions of these models. We will see how to describe neural systems using differential equations, how model simplifications can be made whilst retaining essential features and how we can understand solutions both through simulation and using techniques from dynamical systems theory. Along the way we will review any necessary concepts from linear algebra and vector calculus.

2.1 The Hodgkin-Huxley model

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 initiated 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).

$$\begin{aligned}
C\frac{\mathrm{d}V}{\mathrm{d}t} &= I - g_{Na}m^3h(V - V_{Na}) - g_K n^4(V - V_K) - g_l(V - V_l)\\
\tau_n(V)\frac{\mathrm{d}n}{\mathrm{d}t} &= n_\infty(V) - n\\
\tau_m(V)\frac{\mathrm{d}m}{\mathrm{d}t} &= m_\infty(V) - m\\
\tau_h(V)\frac{\mathrm{d}h}{\mathrm{d}t} &= h_\infty(V) - h
\end{aligned}$$

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

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

3 Differential equations

A differential equation is a relationship between one or more unknown functions and their derivatives. The functions represent physical quantities and the derivatives represent their rates of change.

3.1 Definitions

If we consider a single quantity $u(t)$ that depends only on time t then its evolution is governed by an **ordinary differential equation** (ODE). Here t is an **independent variable** and u is the **dependent variable**. If u depends two or more variables, say $u(x,t)$ then we have a **partial differential equation** (PDE).

The **order** of a differential equation is the order of the highest derivative in the equation.

A differential equation of the dependent variable u is said to be **linear** if the only u-dependent terms are u itself and derivatives of u and also u and its derivatives do not appear multiplied together. If a differential equation does not satisfy these conditions, it is said to be **nonlinear**. Typically, nonlinear equations are harder to solve than linear ones, but exhibit a much greater variety of behaviour.

A differential equation is **autonomous** if it does not depend explicitly on the independent variables.

3.2 Examples

$$\frac{\mathrm{d}N}{\mathrm{d}t} = aN \tag{1}$$

$$C\frac{\mathrm{d}V}{\mathrm{d}t} = V^2 + I \tag{2}$$

$$\frac{\partial V}{\partial t} = -\frac{V}{\tau} + D\frac{\partial^2 V}{\partial x^2} + A(x,t) \tag{3}$$

$$\begin{cases} \epsilon\dfrac{\mathrm{d}V}{\mathrm{d}t} &= V(a-V)(V-1) - W + I \\ \dfrac{\mathrm{d}W}{\mathrm{d}t} &= \beta V - W \end{cases} \tag{4}$$

Let's think about example (1) in more detail to get an idea of the behaviour of its solutions:

Consider the expression for unrestricted population growth

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

where $N = N(t)$ 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. Assuming a positive initial population $N(0) > 0$, if a is positive, our rate of change $\frac{\mathrm{d}N}{\mathrm{d}t}$ will be positive, i.e. the population size is going to increase. 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?

3.3 Solving differential equations analytically

The **state** of the system at time t is the value of all dependent variables at that time. We want to know how the state evolves from a given initial state (this is the **solution** of the system). All of the equations above are **deterministic** so that, given the initial state of the system, the differential equation determines the state of the system at all later times.

Sometimes differential equations can be solved to find an expression for the state variables as functions of time:

Recall our differential equation for unrestricted population growth

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

With initial population $N(0) = N_0$, this equation can be solved analytically by separating the derivative $\frac{\mathrm{d}N}{\mathrm{d}t}$ and integrating both sides:

$$\begin{aligned}\int_{N_0}^{N} \frac{1}{N'}\mathrm{d}N' &= \int_0^t a\,\mathrm{d}t' \\ \log(N) - \log(N_0) &= at \\ N(t) &= N_0 \mathrm{e}^{at}\end{aligned}$$

See Ordinary differential equation examples on Maths Insight for details on how to solve certain classes of 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:

```
[ ]: %matplotlib inline
     import numpy as np
     import matplotlib.pyplot as plt
     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([0,25,0,120000])
     plt.show()
```

Unfortunately, most ODEs do not have explicit expressions for their solutions, and we are forced to rely on other methods to determine how the solutions behave.

- Solutions can be approximated numerically
- The long term qualitative behaviour of solutions can be determined using dynamical systems theory. (For a given initial state, does the system state decay to zero, grow indefinitely, grow to a finite value, or oscillate in time?)

We will introduce both approaches here, but both need some background in linear algebra: to study differential equations numerically, we need to manipulate and store arrays of numbers and to study systems of differential equations we need to keep track of the dependent variables in arrays since for example the state space for HH is $(V(t), n(t), m(t), h(t))$ which is a time-dependent vector.

4 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 blocks of linear algebra are vectors and matrices.

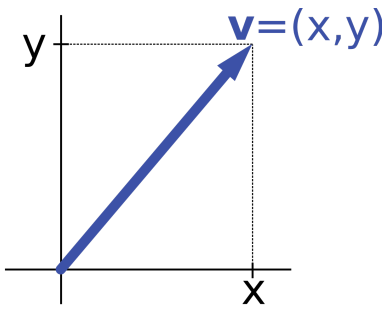
In this lecture will only cover the basics of linear algebra, if you would like to learn more, the Khan Academy course on linear algebra is a good place to start.

4.1 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]
```

4.1.1 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.

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

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

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

by hand and then use Python to check your answer.

```
[ ]: a = np.array([5,1,-9,3,7])
     b = np.array([7,3,1,-4,6])
     a+b
```

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.

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

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

```
[ ]: u*v
```

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?

```
[ ]: a.dot(b)
```

4.2 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')
     plt.show()
```

Transpose The transpose of a matrix A is the matrix A^T with the rows and columns of A swapped. For example

$$A = \begin{bmatrix} 2 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 1 & 6 \\ 8 & 3 & 5 \end{bmatrix}, \qquad A^T = \begin{bmatrix} 2 & 1 & 5 & 8 \\ 8 & 0 & 1 & 3 \\ 4 & 3 & 6 & 5 \end{bmatrix}.$$

The transpose of a $n \times m$ matrix is an $m \times n$ matrix.

```
[ ]: np.transpose(B)
```

The transpose of a column vector v is a row vector v^T. If

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

then

$$v^T = \begin{bmatrix} x & y & z \end{bmatrix}.$$

Note that Python returns the same array for the transpose of a vector:

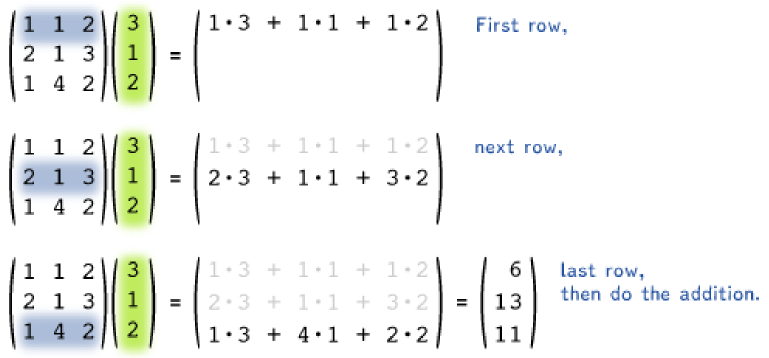
```
[ ]: np.transpose(v)
```

4.2.1 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.

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

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],[1]])
B.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(B)
```

Exercise 4: Compute the product Mu, where

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

by hand and then use Python to check your answer.

[ ]:
```
M = np.array([[1,7,3],[9,6,7]])
u = np.array([[2],[9], [6]])
M.dot(u)
```

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.

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.

```
[ ]: X = np.array([[5,1],[6,3]])
     Y = np.array([[3,7,2],[1,4,6]])
     X.dot(Y)
```

4.2.2 Square Matrices

Square matrices have the same number of rows as columns. An important square matrix is the $n \times n$ identity matrix I_n which has ones on the diagonal and zeros off the diagonal.

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The identity matrix has the property that $PI_n = P = I_nP$ for any $n \times n$ square matrix P and $I_nv = v$ for any column vector v of dimension n.

For square matrices P and Q of the same dimension PQ and QP both exist but $PQ \neq QP$ in general.

Exercise 6: Compute and compare the products PQ and QP where

$$P = \begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix} \quad \text{and} \quad Q \begin{bmatrix} -2 & 1 \\ 4 & 6 \end{bmatrix}.$$

```
[ ]: P=np.array([[1, 2], [2, 2]])
     Q=np.array([[-2, 1], [4, 6]])
     print("PQ=",  P.dot(Q))
     print('QP=', Q.dot(P))
```

Determinants The determinant of a square matrix, A, is a scalar (number) $\det(A)$ that depends on the entries. For a 2×2 matrix

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad \det(A) = a_{11}a_{22} - a_{12}a_{21}.$$

For a 3×3 matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix},$$

$$\begin{aligned}\det(A) &= a_{11} \det\left(\begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix}\right) - a_{12} \det\left(\begin{bmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{bmatrix}\right) + a_{13} \det\left(\begin{bmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}\right) \\ &= a_{11}(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{23}a_{31}) + a_{13}(a_{21}a_{32} - a_{22}a_{31}).\end{aligned}$$

The form of the determinant of an $n \times n$ matrix can be found on Wikipedia (determinants). Python can compute determinants for you.

```
[ ]: np.linalg.det(P)
```

```
[ ]: Q=np.array([[-2, 1, 3, -4], [4, 6, 3, 2], [1, -1, 0, 4], [0, 1, 5, 2]])
     round(np.linalg.det(Q))
```

Inverses If an $n \times n$ square matrix A has $\det(A) \neq 0$ then A has an inverse matrix A^{-1} satisfying $AA^{-1} = A^{-1}A = I_n$. If $\det(A) = 0$ then A is singular and has no inverse. For a 2×2 matrix A, if $\det(A) \neq 0$ then

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad A^{-1} = \frac{1}{\det A} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}.$$

Suppose that $Av = b$ for a given invertible square matrix A and vector b and that the vector v is unknown. Then multiplying both sides on the left by A^{-1} we have

$$v = A^{-1}b.$$

There is a way to compute inverse matrices in higher dimensions by hand: see Wikipedia (Invertible matrix), but it is tedious. Python can calculate inverse matrices

```
[ ]: np.linalg.inv(P)
```

```
[ ]: np.linalg.inv(Q)
```

Eigenvalues and eigenvectors Let M be an $n \times n$ square matrix. If $Mv = \lambda v$ for some nonzero vector v and scalar λ then we say that λ is an eigenvalue of M with corresponding eigenvector v.

For example

$$\begin{bmatrix} 1 & 5 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 6 \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

therefore 6 is an eigenvalue of the matrix with eigenvector $\begin{bmatrix} 1 & 1 \end{bmatrix}^T$.

If $Mv = \lambda v$ then

$$Mv - \lambda v = 0 \qquad \Rightarrow \qquad (M - \lambda I_n)v = 0.$$

If we multiply on the left by $(M - \lambda I_n)^{-1}$ then we see that $v = 0$ which is excluded. Therefore we conclude that $(M - \lambda I_n)^{-1}$ does not exist. That is, $(M - \lambda I_n)$ is singular so $\det(M - \lambda I_n) = 0$. Therefore the eigenvalues of M are the values λ such that $\det(M - \lambda I_n) = 0$ which can be used to compute the eigenvalues as the roots of a polynomial of degree n called the characteristic polynomial.

For example if $M = \begin{bmatrix} 1 & 5 \\ 2 & 4 \end{bmatrix}$ then

$$\begin{aligned} \det(M - \lambda I_2) &= \det\left(\begin{bmatrix} 1-\lambda & 5 \\ 2 & 4-\lambda \end{bmatrix}\right) \\ &= (1-\lambda)(4-\lambda) - 10 \\ &= \lambda^2 - 5\lambda - 6 \\ &= (\lambda - 6)(\lambda + 1). \end{aligned}$$

This is zero when $\lambda = 6$ or $\lambda = -1$ so these are the eigenvalues of M

Exercise 7: Calculate the eigenvalues of

$$M = \begin{bmatrix} 5 & 2 \\ 2 & 5 \end{bmatrix}$$

by hand and then check your answer by running the code below.

```
[ ]: M=np.array([[5,2], [2,5]])
     np.linalg.eig(M)
```

The first array shows the eigenvalues and the corresponding eigenvectors are the columns of the second array. Note that eigenvectors v can also be computed by hand, but we won't do that here. See Wikipedia (Eigenvalues and Eigenvectors) for details.

5 Solving differential equations numerically

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.

We begin by looking at how to determine numerical estimates for solutions to ordinary differential equations. See the Wikipedia page Numerical_methods_for_ordinary_differential_equation for an overview of the different numerical methods for solving ODEs. We will give a brief summary of a couple of the simplest or most commonly used methods. In practice, you will likely use a built in ODE solver.

5.1 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.

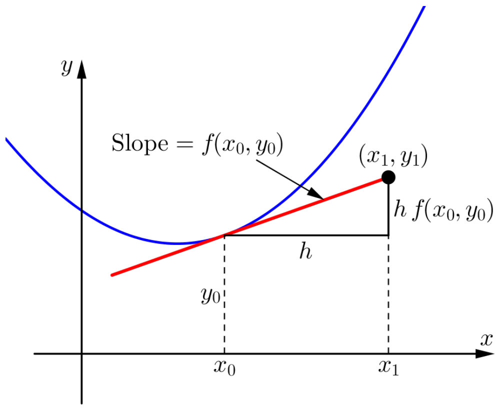
To approximately solve a differential equation

$$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y)$$

with $y(x_0) = y_0$ and step size h, approximate the value of y at $x_1 = x_0 + h$ by $y_1 = y_0 + hf(x_0, y_0)$ where $f(x_0, y_0)$ is the derivative of y at (x_0, y_0). Repeating this for further steps

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

is the approximate value of $y(x_0 + nh)$.



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 differential 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.

```
[ ]: N0 = 10000
     a = 0.1
     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])


     print(N_est)
```

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()
     plt.show()
```

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

[ ]:
```
dt = 0.1
t_est2 = np.arange(0,25+dt,dt)
N_est2 = np.zeros(t_est2.size)
N_est2[0] = N0
for j in range(len(t_est2)-1):
    N_est2[j+1] = N_est2[j] + dt*dNdt(a,N_est2[j])

plt.figure()
plt.plot(t_est2,N_est2,label='Estimate')
plt.plot(t,N,label='Exact')
plt.xlabel('Time (years)')
plt.ylabel('Population size')
plt.axis([0,25,N0,121000])
plt.legend()
plt.show()
```

5.2 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:

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

where

$$\begin{aligned}
k_1 &= \Delta t f(x_n) \\
k_2 &= \Delta t f(x_n + \tfrac{1}{2}k_1) \\
k_3 &= \Delta t f(x_n + \tfrac{1}{2}k_2) \\
k_4 &= \Delta t f(x_n + k_3)
\end{aligned}$$

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
t_RK = np.arange(0,25+dt,dt)
N_RK = np.zeros(t_RK.size)
N_RK[0] = N0
for n in range(len(t_RK)-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_RK,N_RK,label='Runge Kutta')

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

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.

5.3 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,N,a):
    return a*N
```

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, 501)
     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_RK,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_RK,N_RK,label='Runge Kutta')
     plt.plot(t_ivp,N_ivp[0],label='Built-in')
     plt.legend()

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

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

Exercise 9: 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 $x(t)$ on the interval $0 \leq t \leq 10$ and initial value $x(0) = 0.5$.

```
[ ]:
```

5.4 Simulating the Hodgkin-Huxley (HH) model

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

$$\begin{aligned}
C\frac{\mathrm{d}V}{\mathrm{d}t} &= I - \overline{g}_{\mathrm{Na}}m^3h(V - V_{\mathrm{Na}}) - \overline{g}_{\mathrm{K}}n^4(V - V_{\mathrm{K}}) - \overline{g}_l(V - V_l) \\
\frac{\mathrm{d}n}{\mathrm{d}t} &= \alpha_n(V)(1-n) - \beta_n(V)n \\
\frac{\mathrm{d}m}{\mathrm{d}t} &= \alpha_m(V)(1-m) - \beta_m(V)m \\
\frac{\mathrm{d}h}{\mathrm{d}t} &= \alpha_h(V)(1-h) - \beta_h(V)h
\end{aligned}$$

These are same as the form of the equations earlier with $X_\infty(V) = \alpha_X(V)\tau_X(V)$, $\tau_X(V) = (\alpha_X(V) + \beta_X(V))^{-1}$ for $X \in \{n, m, h\}$.

5.4.1 Aside: What do all of the terms in the equations mean and where do they come from?

Differences in ion concentrations inside and outside of the neuron create an electrical potential difference called the membrane potential ($V(t)$). Here the neuron is considered to have the same membrane potential $V(t)$ everywhere across the cell (isopotential). The constant that describes the relationship between the voltage and charge $Q(t)$ that builds up is called the capacitance (C). Differentiating the identity $Q = CV$, $C\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{\mathrm{d}Q}{\mathrm{d}t} = I(t)$ where $I(t)$ is current. We then have, by conservation of electric charge,

$$C\frac{\mathrm{d}V}{\mathrm{d}t} = -I_{\mathrm{ion}} + I,$$

where I is applied current and I_{ion} is the sum of individual ionic currents. HH considers three currents (Na^+, K^+ and a leak current). Each ionic current has a reversal potential (the value of V at which there is no flow of ions across the membrane, V_{K}, V_{Na}, V_l). Recalling also that $I = V/R$ and the reciprocal of resistance R is conductance,

$$I_{\mathrm{ion}} = g_{\mathrm{K}}(V - V_{\mathrm{K}}) + g_{\mathrm{Na}}(V - V_{\mathrm{Na}}) + g_l(V - V_l),$$

where g_{K}, g_{Na}, g_l are conductances. The leak conductance is constant $g_l = \overline{g}_l$, whilst g_{K} and g_{Na} depend on the gating variables n, m, h. This dependence was originally chosen to fit the experimental data but it was subsequently found that potassium conductance depends on four independent activation gates: $g_{\mathrm{K}} = \overline{g}_{\mathrm{K}}n^4$ and sodium conductance depends on three independent activation gates and one inactivation gate: $g_{\mathrm{Na}} = \overline{g}_{\mathrm{Na}}m^3h$. Here $\overline{g}_{\mathrm{K}}$ and $\overline{g}_{\mathrm{Na}}$ are constant maximal conducatances and n, m, h are gating variables which take values between 0 and 1 with dynamics depending on V as in the HH equations above. Roughly speaking they describe the probability that a gate is open and that probability changes with voltage. The $\alpha_X(V)$ and $\beta_X(V)$ are transcendental functions of V chosen to fit experimental data.

5.4.2 Now simulate the HH equations

First, we set up a function that takes the variables (V, n, m, h) as a vector and returns the right-hand side of each of the equations, again as a vector. We will vary the input current I and therefore 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 = 0.01*(V+55)/(1-np.exp(-0.1*(V+55)))
         alpha_m = 0.1*(V+40)/(1-np.exp(-0.1*(V+40)))
         alpha_h = 0.07*np.exp(-0.05*(V+65))

         beta_n = 0.125*np.exp(-0.0125*(V+65))
         beta_m = 4*np.exp(-0.0556*(V+65))
         beta_h = 1/(1+np.exp(-0.1*(V+35)))


         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. In this simulation we set the external inout current $I = 0$ and give an initial perturbation to the initial voltage to raise it to -50 mV.

```
[ ]: # Define parameters
     C = 1
     gNa = 120
     gK = 36
     gL = 0.3
     VNa = 50
     VK = -77
     VL = -54.402

     I=0
     # Simulate model
     HH_sol = solve_ivp(HH, [0,20], [-50,0,0,0.6], dense_output = True, args = (I,))
     t_HH = np.linspace(0, 20, 10000)
     x_HH = HH_sol.sol(t_HH)

     # plot all variables
     fig, ax1 = plt.subplots()

     color = 'tab:blue'
     ax1.set_xlabel('time (ms)')
```

```
ax1.set_ylabel('Voltage (mV)', color=color)
ax1.plot(t_HH, x_HH[0], color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()

color = 'tab:gray'
ax2.plot(t_HH, x_HH[1], color=color, linestyle='dotted', label='n')
ax2.plot(t_HH, x_HH[2], color=color, linestyle='dashed', label='m')
ax2.plot(t_HH, x_HH[3], color=color, linestyle='solid', label='h')
ax2.tick_params(axis='y', labelcolor=color)

plt.legend()
plt.show()
```

Exercise 10: Vary the current I (the input argument) and describe how the dynamics of the voltage change as I is increased. Now plot only the voltage against time and increase the length of the simulation to 100 ms.

```
[ ]: # Simulate model
I=4

HH_sol = solve_ivp(HH, [0,100], [-60,0,0,0.6], dense_output = True, args = (I,))
t_HH = np.linspace(0, 100, 10000)
x_HH = HH_sol.sol(t_HH)

plt.plot(t_HH, x_HH[0])
plt.xlabel('Time')
plt.ylabel('Voltage')
plt.show()
```

6 Qualitative Analysis of Ordinary Differential Equations

Numerical simulations can only tell us an approximation of the solution to a system of ODEs for the initial condition and parameters we set. How can we find out more about the possible solutions for different parameter values and initial conditions without carrying out exhaustive numerical simulations? Is there another way to get some deeper insights into the observed behaviour of a model? For example, how and why does the behaviour of solutions to HH for the parameters we have chosen change from a single action potential to repetative firing as I is varied? We will look to understand this behaviour by considering lower dimensional models.

The four dimensional Hodgkin-Huxley equations include a lot of detail and are able to account for spike shape due to the dependence on the three gating variables. However, when considering networks consisting of large numbers of coupled neurons, analysis is extremely difficult when each element has complicated dynamics. Reduced models capturing the qualitative characteristics of spike generation are preferable. There are various two-dimensional neuron models which can be considered reductions of Hodgkin-Huxley.

- We can consider a model where the variable m is taken to be constant and n and $1-h$ are approximated by a single effective variable w. See Reduction to 2 dimensions.
- The FitzHugh-Nagumo model is an idealised two-variable model that is widely studied as a qualitative prototype for excitable systems.
- The Morris-Lecar model is a two-dimensional system of ODEs incorporating potassium and calcium currents along with the leak current. It assumes that the calcium current operates on a much faster timescale than the potassium current and so it is taken to be instantaneous.

We will consider how the dynamics of the Morris-Lecar model can be understood using phase plane analysis.

6.1 Understanding dynamics using phase plane analysis

Here we will study the qualitative behaviour of solutions of the Morris-Lecar equations:

$$\begin{aligned} C\frac{\mathrm{d}V}{\mathrm{d}t} &= I - g_L(V-V_L) - g_K w(V-V_K) - g_{Ca}m_\infty(V)(V-V_{Ca}) \\ \frac{\mathrm{d}w}{\mathrm{d}t} &= \phi(w_\infty(V)-w)/\tau_w(V) \end{aligned}$$

where

$$\begin{aligned} m_\infty(V) &= 0.5(1+\tanh((V-V_1)/V_2)) \\ w_\infty(V) &= 0.5(1+\tanh((V-V_3)/V_4)) \\ 1/\tau_w(V) &= \cosh((V-V_3)/2V_4) \end{aligned}$$

and V_1, V_2, V_3, V_4 and ϕ are constants given by $V_1 = -1.2$ mV, $V_2 = 18$ mV, $V_3 = 2$ mV, $V_4 = 30$ mV, $\phi = 0.04$ ms^{-1}, and $V_K = -84$ mV, $V_L = -60$ mV, $V_{Ca} = 120$ mV, $g_K = 8$ mmho cm^{-2}, $g_L = 2$ mmho cm^{-2}, $g_{Ca} = 4.4$ mmho cm^{-2} and $C = 20\mu$F cm^{-2}.

We let

$$F(V,w) = -g_L(V-V_L) - g_K w(V-V_K) - g_{Ca}m_\infty(V)(V-V_{Ca}).$$

Solutions of this system of ODEs are $(V(t), w(t))$ in the (V,w) plane, or phase plane. The solution at each time is a point in the phase plane and this moves around as t increases, tracing out a curve, or trajectory in the phase plane.

Let's plot some numerically approximated solutions in the phase plane:

```
[ ]: # Define the ODEs
     def ML(t,x,I):

         V = x[0]
         w = x[1]

         m_inf= 0.5*(1+np.tanh((V-V1)/V2))
         w_inf= 0.5*(1+np.tanh((V-V3)/V4))
```

```
    tau_w= 1/(np.cosh((V-V3)/(2*V4)))

    dVdt = (1/C)*(I - gCa*m_inf*(V-VCa) - gK*w*(V-VK) - gL*(V-VL))
    dwdt = phi*(w_inf-w)/tau_w

    return [dVdt, dwdt]

# Define parameters
C = 20
gCa = 4.4
gK = 8
gL = 2
VCa = 120
VK = -84
VL = -60
V1=-1.2
V2=18
V3=2
V4=30
phi=0.04
```

[ ]:
```
I=0

# Simulate model for different initial conditions
ML_sol1 = solve_ivp(ML, [0,500], [-40,0.0], dense_output = True, args = (I,))
ML_sol2 = solve_ivp(ML, [0,500], [-20,0.0], dense_output = True, args = (I,))
ML_sol3 = solve_ivp(ML, [0,500], [-15,0.0], dense_output = True, args = (I,))
ML_sol4 = solve_ivp(ML, [0,500], [20,0.0], dense_output = True, args = (I,))
t_ML = np.linspace(0, 500, 10000)
x_ML1 = ML_sol1.sol(t_ML)
x_ML2 = ML_sol2.sol(t_ML)
x_ML3 = ML_sol3.sol(t_ML)
x_ML4 = ML_sol4.sol(t_ML)

# plot variables against time
fig, ax1 = plt.subplots()

color = 'tab:blue'
ax1.set_xlabel('time (ms)')
ax1.set_ylabel('Voltage (mV)', color=color)
ax1.plot(t_ML, x_ML4[0], color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()

color = 'tab:gray'
ax2.plot(t_ML, x_ML3[1], color=color, linestyle='solid')
```

```
ax2.set_ylabel('w', color=color)
ax2.tick_params(axis='y', labelcolor=color)

plt.show()

# plot trajectories in (V, w) plane

plt.figure()
plt.plot( x_ML1[0], x_ML1[1],linewidth=2)
plt.plot( x_ML2[0], x_ML2[1],linewidth=2)
plt.plot( x_ML3[0], x_ML3[1],linewidth=2)
plt.plot( x_ML4[0], x_ML4[1],linewidth=2)
plt.ylabel("Potassium gating variable (w)")
plt.xlabel("membrane voltage V (mV)")
plt.show()
```

The velocity vector of the solution curve at $(V(t_0), w(t_0)) = (V_0, w_0)$ is given by

$$\left(\frac{\mathrm{d}V}{\mathrm{d}t}, \frac{\mathrm{d}w}{\mathrm{d}t}\right) = \left(\frac{I + F(V_0, w_0)}{C}, \frac{\phi(w_\infty(V_0) - w_0)}{\tau_w(V_0)}\right).$$

This vector will point in the direction that the solution is flowing and characterises the solution curves.

We can use python to plot the vector field at a grid of points:

```
[ ]: I=0
def SysML(X, t=0):

    return np.array([ (1/C)*(I - gCa*0.5*(1+np.tanh((X[0]-V1)/V2))*(X[0]-VCa) -␣
 ↪gK*X[1]*(X[0]-VK) - gL*(X[0]-VL)),
                        phi*(0.5*(1+np.tanh((X[0]-V3)/V4))-X[1])/(1/(np.
 ↪cosh((X[0]-V3)/(2*V4)))) ])

plt.figure(figsize=(8,6))

# draw the direction field
# define a grid and compute direction at each point
u = np.linspace(-60, 60, 20)
v = np.linspace(0, 0.5, 20)

U , V  = np.meshgrid(u, v)                    # create a grid
DU, DV = SysML([U, V])                          # compute growth rate on the grid
M = (np.hypot(DU, DV))                        # norm growth rate
M[ M == 0] = 1.                               # avoid zero division errors
#DU /= M                                       # normalize each arrows
#DV /= M

plt.quiver(U, V, DU, DV, pivot='mid')
```

```
#plt.plot( x_ML[0], x_ML[1],linewidth=2)
#plt.ylabel("Potassium gating variable (w)")
#plt.xlabel("membrane voltage V (mV)")
plt.show()
```

Note that the arrows are mostly horizontal since ϕ is small so $\frac{\mathrm{d}w}{\mathrm{d}t}$ is small.

6.1.1 Equilibrium points and nullclines

Equilibrium points of the system

$$
\begin{aligned}
C\frac{\mathrm{d}V}{\mathrm{d}t} &= I + F(V,w)\\
\frac{\mathrm{d}w}{\mathrm{d}t} &= \phi(w_\infty(V) - w)/\tau_w(V)
\end{aligned}
$$

are the rest states where $\frac{\mathrm{d}V}{\mathrm{d}t} = 0$ and $\frac{\mathrm{d}w}{\mathrm{d}t} = 0$ so they satsify

$$I + F(V,w) = 0 \quad \text{and} \quad w = w_\infty(V).$$

These two curves in the (V,w) plane are called the V- and w-**nullclines** respectively and equilibrium points occur where they intersect. There can be more than one intersection (more than one rest state) depending on parameter values. Equilibrium points are also often called **steady states** or **fixed points**.

Let's plot the nullclines for $I = 0$ along with the numerical solutions

[ ]:
```
plt.figure()
plt.plot( x_ML1[0], x_ML1[1],linewidth=2)
plt.plot( x_ML2[0], x_ML2[1],linewidth=2)
plt.plot( x_ML3[0], x_ML3[1],linewidth=2)
plt.plot( x_ML4[0], x_ML4[1],linewidth=2)

plt.ylabel("Potassium gating variable (w)")
plt.xlabel("membrane voltage V (mV)")
#generate nullclines

delta = 0.025
xrange = np.arange(-80, 60, delta)
yrange = np.arange(-0.1, 0.5, delta)
Vn, wn = np.meshgrid(xrange,yrange)

F = (1/C)*(I - gCa*0.5*(1+np.tanh((Vn-V1)/V2))*(Vn-VCa) - gK*wn*(Vn-VK) -␣
↪gL*(Vn-VL));
G =  phi*(0.5*(1+np.tanh((Vn-V3)/V4))-wn)/(1/(np.cosh((Vn-V3)/(2*V4))));

#plot nullclines
plt.contour(Vn, wn, F, [0])
plt.contour(Vn, wn, G, [0])
plt.show()
```

Note that the trajectories cross the V nullcline vertically and the w nullcline horizontally. For this choice of parameter values there is a single equilibrium point $(\overline{V}, \overline{w})$. The N shaped V nullcline moves up and down as we change I, moving the location of the equilibrium point so that $(\overline{V}(I), \overline{w}(I))$. The numerical solutions suggest that when $I = 0$, the equilibrium point is **asymptotically stable**. That is, for any nearby initial condition, the solution tends to $(\overline{V}(0), \overline{w}(0))$ as $t \to \infty$. How does the stability of $(\overline{V}(I), \overline{w}(I))$ change as I is increased?

6.1.2 Stability of equilibrium points in a general planar system

Suppose that

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(x, y), \qquad \frac{\mathrm{d}x}{\mathrm{d}t} = g(x, y)$$

has an equilibrium point at $(\overline{x}, \overline{y})$ so that $f(\overline{x}, \overline{y}) = 0$ and $g(\overline{x}, \overline{y}) = 0$. Make a small perturbation away from the equilibrium point so that $x = \overline{x} + u$, $y = \overline{y} + v$. Then Taylor expanding

$$\begin{aligned}
\frac{\mathrm{d}u}{\mathrm{d}t} &= f(\overline{x}+u, \overline{y}+v) \approx f(\overline{x}, \overline{y}) + \frac{\partial f}{\partial x}(\overline{x}, \overline{y})u + \frac{\partial f}{\partial y}(\overline{x}, \overline{y})v + \ldots \\
\frac{\mathrm{d}v}{\mathrm{d}t} &= g(\overline{x}+u, \overline{y}+v) \approx g(\overline{x}, \overline{y}) + \frac{\partial g}{\partial x}(\overline{x}, \overline{y})u + \frac{\partial g}{\partial y}(\overline{x}, \overline{y})v + \ldots.
\end{aligned}$$

Writing $\mathbf{u} = (u, v)^T$ we have the linear equation

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = J\mathbf{u} \quad \text{where} \quad J = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix}_{(\overline{x}, \overline{y})}.$$

The matrix of partial derivatives J is called the **Jacobian**.

Looking for solutions of the form $\mathbf{u} = \mathrm{e}^{\lambda t}\mathbf{u}_0$ we see that

$$\lambda \mathbf{u}_0 = J\mathbf{u}_0$$

so that λ is an eigenvalue of J with corresponding eigenvector $\mathbf{u}_0$. The perturbation $\mathbf{u}$ grows in the direction of $\mathbf{u}_0$ if $\mathrm{Re}(\lambda) > 0$ and decays if $\mathrm{Re}(\lambda) < 0$. In general J has two eigenvalues $\lambda_{1,2}$ that are the roots of the quadratic

$$\lambda^2 - \mathrm{Trace}(J)\lambda + \det(J) = 0$$

where

$$\mathrm{Trace}(J) = \frac{\partial f}{\partial x}(\overline{x}, \overline{y}) + \frac{\partial g}{\partial y}(\overline{x}, \overline{y}), \quad \det(J) = \frac{\partial f}{\partial x}(\overline{x}, \overline{y})\frac{\partial g}{\partial y}(\overline{x}, \overline{y}) - \frac{\partial f}{\partial y}(\overline{x}, \overline{y})\frac{\partial g}{\partial x}(\overline{x}, \overline{y}).$$

We say that $(\overline{x}, \overline{y})$ is asymptotically stable if all perturbations decay as $t \to \infty$ i.e., if $\lambda_{1,2}$ both have $\mathrm{Re}(\lambda_{1,2}) < 0$. This occurs when $\mathrm{Trace}(J) < 0$ and $\det(J) > 0$. There are two possible ways for a stable equilibrium point to lose stability 1. A real eigenvalue passes through zero when $\det(J) = 0$ 2. A pair of complex conjugate eigenvalues have real part passing through zero when $\mathrm{Trace}(J) = 0$ and $\det(J) > 0$.

Case 1 corresponds to a saddle node bifurcation. Case 2 corresponds to a Hopf bifurcation that leads to periodic solutions.

6.1.3 Oscillations emerging with non-zero frequency (Hopf bifurcation)

We now return to the Morris-Lecar model and investigate instabilities of the steady states looking for the onset of periodic solutions (oscillations). These are solutions where $(V(t+T), w(t+T)) = (V(t), w(t))$ for $T > 0$ and they appear as closed loops in the phase plane diagram.

Assuming that $\tau_w(V)$ is only slowly varying so can be considered constant, the Jacobian at a steady state $(\overline{V}(I), \overline{w}(I))$ is given by

$$J = \begin{bmatrix} \frac{1}{C}\frac{\partial F}{\partial V} & \frac{1}{C}\frac{\partial F}{\partial w} \\ \frac{\phi}{\tau_w}\frac{\partial w_\infty}{\partial V} & -\frac{\phi}{\tau_w} \end{bmatrix}_{(\overline{V}(I), \overline{w}(I))}.$$

The equilibrium points $(\overline{V}(I), \overline{w}(I))$ satisfy $\overline{w}(I) = w_\infty(\overline{V}(I))$, where

$$F(\overline{V}(I), w_\infty(\overline{V}(I))) + I = 0.$$

Plotting $-F(V, w_\infty(V))$ for our parameter values we see that it is always increasing with increasing V and so $\frac{\mathrm{d}F}{\mathrm{d}V}(\overline{V}, \overline{w}) < 0$ and there is only ever one value of $\overline{V}(I)$ for each value of I giving a single equilibrium point for our parameter values. Here $I_{ss}(V) = -F(V, w_\infty(V))$ is the unique value of I which puts the equilibrium point at V.

```
[ ]: Vss = np.linspace(-60,40,1000)
     F =  gCa*0.5*(1+np.tanh((Vss-V1)/V2))*(Vss-VCa) + gK*0.5*(1+np.tanh((Vss-V3)/
      ↪V4))*(Vss-VK) + gL*(Vss-VL)
     plt.figure()
     plt.plot(Vss,F)
     plt.xlabel('V')
     plt.ylabel('$I_{ss} = -F(V, w_\infty(V))$')
     plt.show()
```

Since

$$\det(J) = -\frac{\phi}{C\tau_w}\left(\frac{\partial F}{\partial V} + \frac{\partial F}{\partial w}\frac{\partial w_\infty}{\partial V}\right) = \frac{\phi}{C\tau_w}\frac{\mathrm{d}I_{ss}}{\mathrm{d}V}$$

and I_{ss} montonically increases with V, there are no values of I for which case 1 can occur since $\det(J) > 0$ for all values of I.

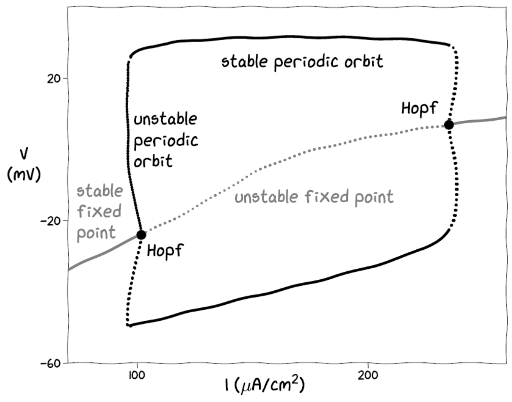
Therefore, destabilisation of the equilibrium point can only occur through case 2, a Hopf bifurcation at a critical value of $I = I_c$, giving rise to a periodic solution. A Hopf bifurcation occurs when $\mathrm{Trace}(J) = 0$, so when

$$\frac{1}{C}\frac{\partial F}{\partial V}(\overline{V}, \overline{w}) = \frac{\phi}{\tau_w}.$$

Since $F(V, W) = 0$ has a cubic like shape for our parameters (see the V-nullcline above), $\frac{\partial F}{\partial V}$ is approximately quadratic in V and hence the condition for a Hopf bifurcation should either have two or no roots $\overline{V}$ giving either two or no values of I_c at which Hopf bifurcations occur, depending on the value of $\frac{C\phi}{\tau_w}$ (due to one-to-one correspondance between $\overline{V}$ and I).

Bifurcation diagrams depict the equilibrium (or steady-state) and periodic solutions which exist over a range of the parameter to be varied (here I). Solid lines indicate stable solutions while dotted lines indicated unstable solutions. The figure below shows a bifurcation diagram with two

Hopf bifurcations. The numerically computed emerging periodic orbits are also shown, indicated by the maximum and minimum values of the oscillating voltage.



Just as equilibrium points can be unstable, so can periodic solutions. The periodic solutions emerging from the Hopf bifurcations in this case are unstable and therefore not observable in numerical simulations. However, they gain stability at turning points. The stable periodic solutions (limit cycles) are repetative firing. Oscillations emerging from Hopf bifurcations have small amplitude and a non-zero emergent frequency proportional to $\text{Im}(\lambda(I_c))$, so there is a jump from zero to a finite firing frequency typically referred to as Type II firing. A Hopf bifurcation is also the mechanism giving rise to oscillations in the Hodgkin-Huxley model we studied above.

Bistability The bifurcation diagram above also shows a (small) range of I values for which both the equilibrium point and the periodic orbit are stable and an unstable periodic orbit also exists. In this case for different initial conditions we see different behaviour. For these values of I a brief impulse could switch the system out of the oscillatory response and back to rest.

Exercise 11: Use the code below to investigate numerically the behaviour for different values of I. It should agree with the bifurcation diagram: For $I = 60$ you should observe only the stable equilibrium point, for $I = 160$ the stable solution is the periodic orbit, for $I = 260$, the stable solution is again just the steady state. For $I = 90$ there is bistability between the equilibrium point and the periodic solution.

[ ]:
```
I=60

# Simulate model for different initial conditions
ML_sol1 = solve_ivp(ML, [0,500], [-40,0.0], dense_output = True, args = (I,))
ML_sol2 = solve_ivp(ML, [0,500], [-20,0.2], dense_output = True, args = (I,))
ML_sol3 = solve_ivp(ML, [0,500], [-15,0.0], dense_output = True, args = (I,))
ML_sol4 = solve_ivp(ML, [0,500], [20,0.0], dense_output = True, args = (I,))
t_ML = np.linspace(0, 500, 10000)
x_ML1 = ML_sol1.sol(t_ML)
x_ML2 = ML_sol2.sol(t_ML)
x_ML3 = ML_sol3.sol(t_ML)
x_ML4 = ML_sol4.sol(t_ML)
```

```
# plot trajectories in (V, w) plane

plt.figure(figsize=(12,8))
plt.plot( x_ML1[0], x_ML1[1],linewidth=2)
plt.plot( x_ML2[0], x_ML2[1],linewidth=2)
plt.plot( x_ML3[0], x_ML3[1],linewidth=2)
plt.plot( x_ML4[0], x_ML4[1],linewidth=2)

delta = 0.025

xrange = np.arange(-80, 60, delta)
yrange = np.arange(-0.1, 0.8, delta)
Vn, wn = np.meshgrid(xrange,yrange)

F = (1/C)*(I - gCa*0.5*(1+np.tanh((Vn-V1)/V2))*(Vn-VCa) - gK*wn*(Vn-VK) -␣
 ↪gL*(Vn-VL));
G =  phi*(0.5*(1+np.tanh((Vn-V3)/V4))-wn)/(1/(np.cosh((Vn-V3)/(2*V4))));

#plot nullclines
plt.contour(Vn, wn, F, [0])
plt.contour(Vn, wn, G, [0])

plt.ylabel("Potassium gating variable (w)")
plt.xlabel("membrane voltage V (mV)")
plt.show()
```

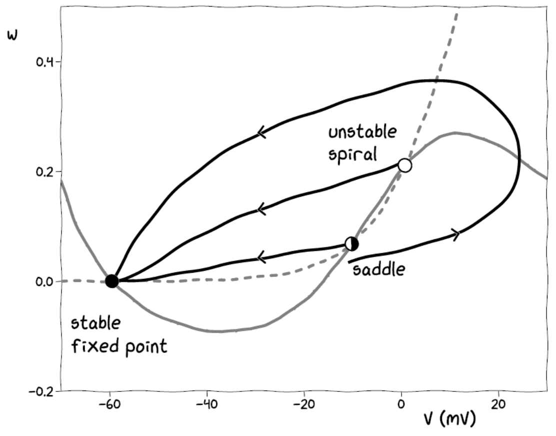
You can use the code below to plot a particular solution against time.

```
[ ]: # plot variables against time
fig, ax1 = plt.subplots()

color = 'tab:blue'
ax1.set_xlabel('time (ms)')
ax1.set_ylabel('Voltage (mV)', color=color)
ax1.plot(t_ML, x_ML2[0], color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()

color = 'tab:gray'
ax2.plot(t_ML, x_ML2[1], color=color, linestyle='solid')
ax2.set_ylabel('w', color=color)
ax2.tick_params(axis='y', labelcolor=color)

plt.show()
```

6.1.4 Oscillations emerging with zero frequency (global bifurcations)

More usual than type II firing is the observation of periodic firing with low frequencies at onset (type I). We now briefly outline two mechanisms for generating such periodic behaviour.

SNIC bifurcation If $F(V, w_\infty(V))$ is non-monotonic (has turning points) then the system can simultaneously have more than one equilibrium point. For example if we change some of the parameters we can get the phase portrait below with three equilibrium points, one stable, one unstable and one saddle. (A saddle point has one positive real eigenvalue and one negative real eigenvalue so trajectories appproach in one direction and move away in the other). There are two trajectories that leave the saddle and both connect to the stable equilibrium point, one goes directly there and one goes around the unstable equilibrium.



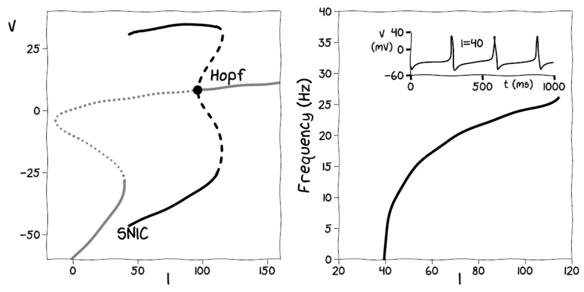
Increasing I moves the V-nullcline upwards, making the stable equilibrium and the saddle closer together until they annihilate each other when $I = I_c$ and then disappear leaving a limit cycle for $I > I_c$. This situation is called a **saddle node on an invariant circle (SNIC) bifurcation**. At $I = I_c$ the limit cycle has infinite period (it is a homoclinic orbit) so the frequency is zero. For I just above I_c the frequency of the periodic orbit scales as $\sqrt{I - I_c}$

Exercise 12: Use the code below to see how the limit cycle comes into existence when the equilibrium points annihilate.

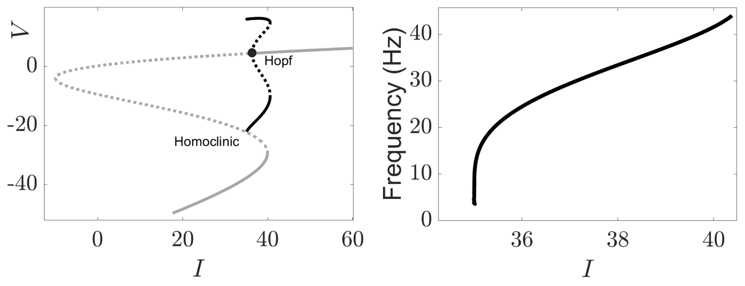
[ ]:
```
# Define new parameter values
V3=12
V4=17
phi = 2/30

# F(V, w_inf(V)) is non-monotonic

Vss = np.linspace(-60,10,1000)
F =  gCa*0.5*(1+np.tanh((Vss-V1)/V2))*(Vss-VCa) + gK*0.5*(1+np.tanh((Vss-V3)/
↪V4))*(Vss-VK) + gL*(Vss-VL)
plt.figure()
plt.plot(Vss,F)
```

```python
plt.xlabel('V')
plt.ylabel('$I_{ss} = -F(V, w_\infty(V))$')
plt.show()

# try I=0, I=39 (just after SNIC), I=90 (bistability), I=120, stable␣
 ↪equilibrium
I=39

# Simulate model for different initial conditions
ML_sol1 = solve_ivp(ML, [0,500], [0,0.3], dense_output = True, args = (I,))
ML_sol2 = solve_ivp(ML, [0,500], [-15,0.03], dense_output = True, args = (I,))
ML_sol3 = solve_ivp(ML, [0,500], [-14,0.03], dense_output = True, args = (I,))
ML_sol4 = solve_ivp(ML, [0,500], [20,0.0], dense_output = True, args = (I,))
t_ML = np.linspace(0, 500, 10000)
x_ML1 = ML_sol1.sol(t_ML)
x_ML2 = ML_sol2.sol(t_ML)
x_ML3 = ML_sol3.sol(t_ML)
x_ML4 = ML_sol4.sol(t_ML)


# plot trajectories in (V, w) plane

plt.figure()
plt.plot( x_ML1[0], x_ML1[1],linewidth=2)
plt.plot( x_ML2[0], x_ML2[1],linewidth=2)
plt.plot( x_ML3[0], x_ML3[1],linewidth=2)
plt.plot( x_ML4[0], x_ML4[1],linewidth=2)

delta = 0.025

xrange = np.arange(-80, 60, delta)
yrange = np.arange(-0.1, 0.8, delta)
Vn, wn = np.meshgrid(xrange,yrange)

F = (1/C)*(I - gCa*0.5*(1+np.tanh((Vn-V1)/V2))*(Vn-VCa) - gK*wn*(Vn-VK) -␣
 ↪gL*(Vn-VL));
G =  phi*(0.5*(1+np.tanh((Vn-V3)/V4))-wn)/(1/(np.cosh((Vn-V3)/(2*V4))));

#plot nullclines
plt.contour(Vn, wn, F, [0])
plt.contour(Vn, wn, G, [0])

plt.ylabel("Potassium gating variable (w)")
plt.xlabel("membrane voltage V (mV)")
plt.show()
```

The corresponding bifurcation diagram is as below.



Homoclinic bifurcation There is another way that oscillations with low frequency at onset can be created in the Morris-Lecar model. For intermediate values of ϕ, it is possible for both the lower and upper equilibrium points to be simultaneously stable for some values of I with the third a saddle point. The upper branch can lose stability through a Hopf bifurcation with decreasing I giving rise to an unstable periodic solution. This turns and become stable and for some window of I there are three steady states (two stable) and two periodic orbits (one stable). Further decreasing I the stable periodic orbit can intercept the saddle, terminating the periodic branch in a **homoclinic bifurcation** at $I = I_c$. At the bifurcation point a trajectory leaves the saddle in its unstable direction and returns along its stable direction. The frequency of oscillations scales as $1/|\ln(I - I_c)|$. The bifurcation diagram is shown below.



Exercise 13: Use the code below to explore the region of bi/tri stability between equilibrium points and the periodic solution between $I = 35$ and $I = 40.6$ for the parameter values given.

```
[ ]: # Define new parameter values
     V3=12
     V4=17.4
     phi=0.23
     gCa=4

     #I=34 # all trajectories go to lower eqm
     #I = 35.5 # bistability between lower eqm and periodic solution
     #I=39 #two eqm and one periodic solution stable
     I=40.3 # bistability between upper eq and periodic solution
     #I=50 # all trajectories go to upper eqm

     # Simulate model for different initial conditions
     ML_sol1 = solve_ivp(ML, [0,1000], [3,0.3], dense_output = True, args = (I,))
     ML_sol2 = solve_ivp(ML, [0,1000], [-20,0.0], dense_output = True, args = (I,))
```

```
ML_sol3 = solve_ivp(ML, [0,1000], [-10,0.0], dense_output = True, args = (I,))
ML_sol4 = solve_ivp(ML, [0,1000], [20,0.0], dense_output = True, args = (I,))
t_ML = np.linspace(0, 1000, 20000)
x_ML1 = ML_sol1.sol(t_ML)
x_ML2 = ML_sol2.sol(t_ML)
x_ML3 = ML_sol3.sol(t_ML)
x_ML4 = ML_sol4.sol(t_ML)


# plot trajectories in (V, w) plane

plt.figure()
plt.plot( x_ML1[0], x_ML1[1],linewidth=2)
plt.plot( x_ML2[0], x_ML2[1],linewidth=2)
plt.plot( x_ML3[0], x_ML3[1],linewidth=2)
plt.plot( x_ML4[0], x_ML4[1],linewidth=2)

delta = 0.025

xrange = np.arange(-80, 60, delta)
yrange = np.arange(-0.1, 0.8, delta)
Vn, wn = np.meshgrid(xrange,yrange)

F = (1/C)*(I - gCa*0.5*(1+np.tanh((Vn-V1)/V2))*(Vn-VCa) - gK*wn*(Vn-VK) -␣
 ↪gL*(Vn-VL));
G =  phi*(0.5*(1+np.tanh((Vn-V3)/V4))-wn)/(1/(np.cosh((Vn-V3)/(2*V4))));

#plot nullclines
plt.contour(Vn, wn, F, [0])
plt.contour(Vn, wn, G, [0])

plt.ylabel("Potassium gating variable (w)")
plt.xlabel("membrane voltage V (mV)")
plt.show()
```

7 Phase reduction

We have looked at how neuron models can respond to constant input, but current inputs in realistic scenarios will be more complex. We now consider how neurons which exhibit oscillations without external input can respond to stimuli, including input from other neurons when coupled together in a network.

7.1 Phase and phase shifts

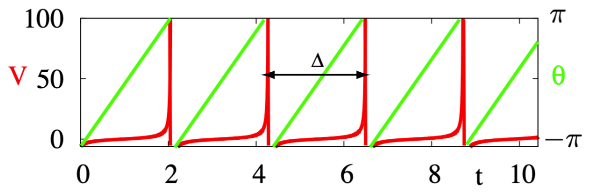
If a periodic oscillation is stable then perturbations return to the oscillation as $t \to \infty$, however the time course may exhibit a time shift from the unperturbed oscillation:

```
[ ]: # Define parameters (as for oscillations through Hopf bifurcation)
C = 20
gCa = 4.4
gK = 8
gL = 2
VCa = 120
VK = -84
VL = -60
V1=-1.2
V2=18
V3=2
V4=30
phi=0.04

I=90

# Simulate model and perturbation
ML_sol1 = solve_ivp(ML, [0,1000], [-40,0.0], dense_output = True, args = (I,))
t_ML = np.linspace(0, 1000, 20000)
x_ML1 = ML_sol1.sol(t_ML)

x0= x_ML1[0][8000]-5
x1= x_ML1[1][8000]
t0= t_ML[8000]
ML_solpert = solve_ivp(ML, [0,400], [x0, x1], dense_output = True, args = (I,))
t_pert = np.linspace(0, 400, 8000)
x_MLpert = ML_solpert.sol(t_pert)

plt.figure()
plt.plot(t_ML[8000:16000], x_ML1[0][8000:16000],linewidth=2, color='tab:gray')
t_MLplus= np.append(t_ML[4000:8000], t0)
x_ML1plus =np.append(x_ML1[0][4000:8000], x0)
plt.plot(t_MLplus, x_ML1plus ,linewidth=2, color= 'tab:blue')
plt.plot(t_pert+t0, x_MLpert[0],linewidth=2, color= 'tab:blue')
plt.xlabel("time")
plt.ylabel("membrane voltage V (mV)")
plt.show()
```

Suppose that the period of an oscillator is Δ and let $t = 0$ correspond to the peak in the voltage so that at $t = \Delta$. We can introduce the notion of the **phase**, $\theta \in [0, 2\pi)$ of the periodic solution. On the periodic orbit, θ increases linearly: $\frac{\mathrm{d}\theta}{\mathrm{d}t} = 2\pi/\Delta = \omega$.

Let $\gamma(\theta)$ be the point on the limit cycle with phase θ. For points x off of the limit cycle we can define an **asymptotic phase**, $\theta(x)$, as the phase along the limit cycle to which the trajectory with initial condition x will tend as $t \to \infty$. The set of all points with the same asymptotic phase define an **isochron**.

We now think about how a perturbation shifts the phase of the oscillator. In the code above, we



apply a brief negative (hyperpolarising) current pulse. We observe that this pulse increases the time to the next peak and we say the phase has been delayed. Let Δ_1 be the time to the next peak. The phase shift is

$$Z(\theta) = \frac{\Delta - \Delta_1(\theta)}{\Delta}.$$

Note that it depends on the phase at which the stimulus is applied. For some values of θ, $Z(\theta)$ may be postive, (stimuli advance the phase, next peak arrives sooner) and for others it may be negative (stimuli delay the phase, next peak occurs later). We call $Z(\theta)$ the **phase response curve** (PRC) and it typically also depends on the form of stimulus (e.g., synaptic or gap junction coupling). PRCs can be inferred from data or other experimental methods. If we know the isochrons then determining the PRC is straightforward (but in general isochrons are difficult to compute). Note that we observed that some models have bistability between oscillations and steady-states. The PRC is not defined for stimuli that push the trajectory to a point where it is then not attracted back to the oscillatory limit cycle.

7.1.1 Infinitesimal phase response curve

Suppose that the (vector) perturbation δx is infinitesimally small. The associated phase shift can be approximated as

$$\delta\theta = \theta(x + \delta x) - \theta(x) = \nabla_x \theta(x) \cdot \delta x.$$

Here $Q = \nabla_x \theta(x)$ is the gradient of the asymptotic phase function and $(\cdot)$ is the dot product. Q is called the infinitesimal phase response function.

Gradient of a function: Let F be a function which takes a vector $\mathbf{x} = (x_1, x_2, \ldots x_n)$ and returns a scalar. Then

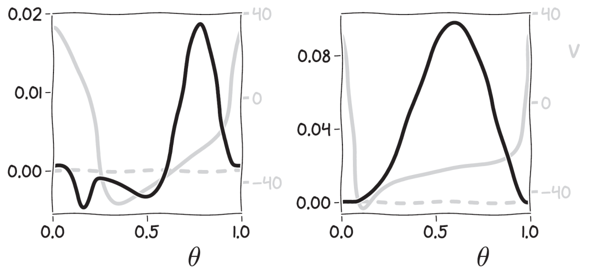
$$\nabla F(\mathbf{x}) = \left(\frac{\partial F}{\partial x_1}, \frac{\partial F}{\partial x_2}, \ldots, \frac{\partial F}{\partial x_n} \right)^T.$$

If the unperturbed trajectory is on the limit cycle and the perturbations are infinitesimally weak, then Q can be evaluated on cycle and can be found numerically by solving the **adjoint** differential equation

$$\frac{\mathrm{d}Q}{\mathrm{d}t} = -J(t)^T Q$$

subject to $Q(t + \Delta) = Q(t)$ and $Q(0) \cdot F(\gamma(0)) = \omega$ where $J(t)$ is the Jacobian evaluated on the limit cycle. XPPAUT can do that for you.

On the left is the first (voltage) component iPRC for Morris-Lecar near a Hopf bifurcation ($I = 100$) and on the right near a SNIC bifurcation ($I = 42$). Usually all perturbations are assumed to be in the voltage variable. All depolarising stimuli near the SNIC lead to advancment of the phase, whereas near the Hopf both phase advances and delays are possible, with advances for depolarising



stimuli applied during the upstroke of the voltage on the limit cycle and delays when stimulus occurs while V is decreasing.

7.1.2 Phase models

If a weak external stimulus is applied to an oscillator

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{F}(\mathbf{x}) + \epsilon \mathbf{s}(t)$$

then it can be described using phase only using the **phase model**

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} = \omega + \epsilon Q(t) \cdot \mathbf{s}(t).$$

7.2 Weakly coupled oscillator networks

Assume that each node in a network has a stable limit cycle in isolation and the coupling is weak so that the trajectories of each node remain close to their limit cycle. Then the dynamics of the network can be captured through looking just at the phase dynamics. Consider a network of N interacting neurons given by

$$\frac{\mathrm{d}\mathbf{x}_i}{\mathrm{d}t} = \mathbf{F}_i(\mathbf{x}_i) + \epsilon \sum_{j=1}^{N} w_{ij} g_{ij}(\mathbf{x}_i(t), \mathbf{x}_j(t))$$

where each isolated neuron possesses an attracting limit cycle γ_i with frequency ω_i. Here g_{ij} describes the form of the coupling between neurons i and j, w_{ij} describes the strength of interaction from node j to i, while ϵ is an overall interaction strength. In phase-reduced form this is

$$\frac{\mathrm{d}\theta_i}{\mathrm{d}t} = \omega_i + \epsilon \sum_{j=1}^{N} w_{ij} Q_i(\theta_i) \cdot g_{ij}(\gamma_i(\theta_i), \gamma_j(\theta_j)). \tag{5}$$

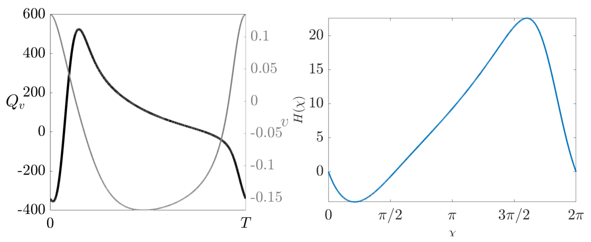
Assuming identical oscillators and coupling $Q_i = Q$ and $g_{ij} = g$, Averaging Theory gives the evolution of the phase up to first order in ϵ as satisfying

$$\frac{\mathrm{d}\theta_i}{\mathrm{d}t} = \omega_i + \epsilon \sum_{j=1}^{N} w_{ij} H(\theta_j - \theta_i)$$

where

$$H(\chi) = \frac{1}{2\pi}\int_0^{2\pi} Q(\phi)\cdot g(\gamma(\phi),\gamma(\phi+\chi)).$$

XPPAUT can also calculate the phase interaction function H for a given coupling function. For linear coupling in the voltage variable $g(\mathbf{x}_i,\mathbf{x}_j) = (v_j - v_i, 0, \ldots, 0)$ (gap junction coupling), the phase interaction function for Morris-Lecar neurons near a homoclinic bifurcation is on the right below with the iPRC on the left.



The celebrated Kuramoto model is an example of this type of system with $w_{ij} = 1/N$ and $H(\chi) = \sin(\chi)$:

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

We can run a simulation of this model with 100 nodes and frequencies ω_i chosen from a distribution.

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

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

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

     import scipy.stats as stats

     # Define parameters
     k = 0.10
     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')
     plt.show()
```

A nicer way to visualise the evolution of the phases is to place them on a circle

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

# 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)

# 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.

7.2.1 Stability of synchrony in weakly coupled networks

The general network equations (5) can be used to study states such as synchrony in networks. Suppose that $w_{ij} = 1/N$, then if $\theta_i = \Omega t$ for a collective frequency Ω, the oscillators are in synchrony. From (5), $\Omega = \omega + H(0)/N$.

The Jacobian is given by $J = -\epsilon \frac{\mathrm{d}H}{\mathrm{d}\chi}(0)L = -\epsilon H'(0)L$ where L is the matrix with $L_{ij} = \delta_{ij} - 1/N$ for $\delta_{ij} = 1$ if $i = j$ and 0 otherwise. This has eigenvalues 1, ($N-1$ times) and 0 (corresponding to perturbations along the periodic orbit). Therefore J has eigenvalues $\lambda = -\epsilon H'(0)$ and synchrony is stable if

$$\epsilon H'(0) > 0.$$

For positive coupling strength ϵ, phase reduction predicts that synchrony is unstable in a network of Morris-Lecar neurons near a homoclinic bifurcation with gap junction coupling since the H above has negative slope at $\chi = 0$.

Does this agree with simulations? Consider a network of two gap junction coupled Morris-Lecar neurons near a homoclinic bifurcation:

[ ]:
```python
# Define the ODEs
def ML2(t,x,eps):

    Va = x[0]
    wa = x[1]
    Vb = x[2]
    wb = x[3]

    m_infa= 0.5*(1+np.tanh((Va-V1)/V2))
    w_infa= 0.5*(1+np.tanh((Va-V3)/V4))
    tau_wa= 1/(np.cosh((Va-V3)/(2*V4)))

    m_infb= 0.5*(1+np.tanh((Vb-V1)/V2))
    w_infb= 0.5*(1+np.tanh((Vb-V3)/V4))
    tau_wb= 1/(np.cosh((Vb-V3)/(2*V4)))

    dVadt = (1/C)*(I - gCa*m_infa*(Va-VCa) - gK*wa*(Va-VK) - gL*(Va-VL)) +␣
↪eps*(Vb-Va)/2
    dwadt = phi*(w_infa-wa)/tau_wa
    dVbdt = (1/C)*(I - gCa*m_infb*(Vb-VCa) - gK*wb*(Vb-VK) - gL*(Vb-VL)) +␣
↪eps*(Va-Vb)/2
    dwbdt = phi*(w_infb-wb)/tau_wb

    return [dVadt, dwadt, dVbdt, dwbdt]

# Define parameters
C = 1
gCa = 1
gK = 2
gL = 0.5
```

```
VCa = 1
VK = -0.7
VL = -0.5
V1=-0.01
V2=0.15
V3=0.1
V4=0.145
phi=1.15
I = 0.075
```

[ ]:
```
eps=0.2
#eps =0.8

# Simulate model for different initial conditions
ML2_sol1 = solve_ivp(ML2, [0,5000], [-0.02,0.24,-0.05,0.25], dense_output =␣
 ↪True, args = (eps,))
t_ML = np.linspace(0, 5000, 100000)
x_ML21 = ML2_sol1.sol(t_ML)

plt.figure(figsize=(8,6))
plt.plot( t_ML[98000:], x_ML21[0][98000:],linewidth=2)
plt.plot( t_ML[98000:], x_ML21[2][98000:],linewidth=2)

plt.ylabel("membrane voltage V (mV)")
plt.xlabel("time")
plt.show()

# plot trajectories in (V, w) plane

plt.figure(figsize=(8,6))
plt.plot( x_ML21[0][98000:], x_ML21[1][98000:],linewidth=2)
plt.plot( x_ML21[2][98000:], x_ML21[3][98000:],linewidth=2)

delta = 0.025

xrange = np.arange(-0.5, 0.5, delta)
yrange = np.arange(-0.1, 0.5, delta)
Vn, wn = np.meshgrid(xrange,yrange)

F = (1/C)*(I - gCa*0.5*(1+np.tanh((Vn-V1)/V2))*(Vn-VCa) - gK*wn*(Vn-VK) -␣
 ↪gL*(Vn-VL));
G =  phi*(0.5*(1+np.tanh((Vn-V3)/V4))-wn)/(1/(np.cosh((Vn-V3)/(2*V4))));

#plot nullclines
plt.contour(Vn, wn, F, [0])
plt.contour(Vn, wn, G, [0])
plt.ylabel("Potassium gating variable (w)")
plt.xlabel("membrane voltage V (mV)")
plt.show()
```

For ϵ small and positive we see that synchrony is unstable as predicted, but as ϵ increases, synchrony becomes stable. This cannot be predicted using phase reduction.

7.3 Beyond phase reduction

Reduction to a single phase variable for each neuron assumes that following any perturbation the dynamics instantaneously return to the limit cycle. This is a good approximation if the limit cycle is strongly attracting and the perturbation is small (or coupling is weak in a network). When this is not the case, off-cycle dynamics become important so in addition to the phase we also include variables describing a distance to the limit cycle which (in the absence of perturbations) decay exponentially at rates determined by the Floquet exponents $\kappa_i < 0$ of the stable limit cycle:

$$\frac{\mathrm{d}\psi_i}{\mathrm{d}t} = \kappa_i \psi_i$$

We call the ψ_i the amplitudes of the dynamics. Suppose for simplicity that decay in all but one direction is fast enough to be ignored (all but one of the κ_i are large and negative) and focus only the dynamics of the most slowly decaying amplitude ψ. Each point x for which a trajectory starting from that point would decay to the limit cycle can be assigned an amplitude. Points with equal amplitude lie on the same **isostable** and decay to the limit cycle in the same amount of time. On the limit cycle $\psi = 0$.

Just as we defined the (infinitesimal) phase response curve, we can define the (infinitesimal) amplitude response curve. The amplitude response to stimuli is quantified by $I = \nabla_x \psi(x)$, the gradient of the amplitude function. Assuming weak stimuli and evaluating on cycle I we get the infinitesimal amplitude response curve (iARC) which can be found as the solution of the differential equation

$$\frac{\mathrm{d}I}{\mathrm{d}t} = -(J(t) - \kappa I_n)^T I(t),$$

where $I(t + \Delta) = I(t)$ and subject to a normalisation condition.

Then we have the phase-amplitude equations for an oscillator with time dependent stimuli $\epsilon \mathbf{s}(t)$:

$$\begin{aligned}\frac{\mathrm{d}\theta}{\mathrm{d}t} &= \omega + \epsilon Q(t) \cdot \mathbf{s}(t),\\ \frac{\mathrm{d}\psi}{\mathrm{d}t} &= \kappa\psi + \epsilon I(t) \cdot \mathbf{s}(t).\end{aligned}$$

Taking the expansions of phase and amplitude response to quadratic order in ϵ we get terms in the phase equation depending on ψ. Using these to describe coupled oscillator networks with idenitcal nodes we obtain network equations which now have six interaction functions.

$$
\begin{aligned}
\frac{\mathrm{d}\theta_i}{\mathrm{d}t} &= \omega + \epsilon \sum_{j=1}^{N} w_{ij} \left(H_1(\theta_j - \theta_i) + \psi_i H_2(\theta_j - \theta_i) + \psi_j H_3(\theta_j - \theta_i) \right), \\
\frac{\mathrm{d}\psi_i}{\mathrm{d}t} &= \kappa\psi + \epsilon \sum_{j=1}^{N} w_{ij} \left(H_4(\theta_j - \theta_i) + \psi_i H_5(\theta_j - \theta_i) + \psi_j H_6(\theta_j - \theta_i) \right).
\end{aligned}
$$

The stability of synchrony in a globally coupled network $w_{ij} = 1/N$ can be considered in this framework where it is found that stability requires

$$
\kappa + \epsilon(H_5(0) - H_1'(0)) < 0, \quad \text{and} \quad -\epsilon H_1'(0)\left(\kappa + \epsilon H_5(0)\right) + \epsilon^2 H_4'(0) H_2(0) > 0.
$$

From this, for the gap junction coupled pair of Morris-Lecar neurons, we can predict that for some value of $\epsilon > 0$ synchrony will restabilise.

Phase-amplitude coordinates are a promising new framework in which to study reductions of oscillator network dynamics and can capture more of the emergent phenomenon than phase reduction.